Peter, the Language that does not Exist...

Luigi Liquori

Peter, le langage qui n'existe pas...

Peter, the Language that does not Exist...

Version d'été / Summer Version

Habilitation à Diriger les Recherches
Habilitation Thesis

Luigi Liquori
Chargé de Recherche INRIA / INRIA Senior Researcher

Nancy, 6 juillet 2007 / Nancy, July 6, 2007

Un grand merci à tous mes coauteurs...
A massive thanks to all my coauthors...

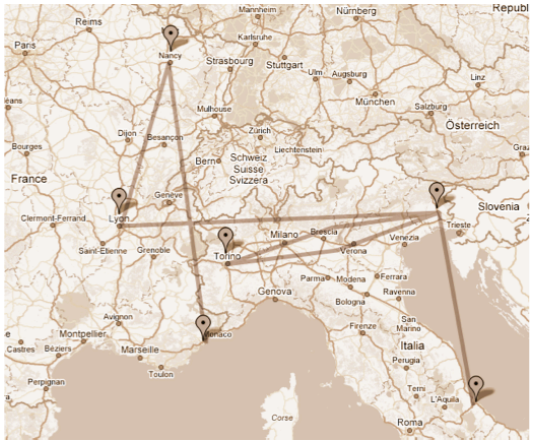
Steffen van Bakel
Gilles Barthe
Didier Benza
Viviana Bono
Michele Bugliesi
Giuseppe Castagna
Raphael Chand
Alberto Ciaffaglione
Horatiu Cirstea
Dominique Colnet
Michel Cosnard
Giorgio Delzanno
Mariangiola Dezani-Ciancaglini
Daniel J. Dougherty
Pietro Di Gianantonio
Furio Honsell
Claude Kirchner
Frédéric Lang
Marina Lenisa
Pierre Lescanne
Maurizio Martelli
Marino Miculan
Rekha Redamalla
Simona Ronchi Della Rocca
Maria Luisa Sapino
Ivan Scagnetto
Bernard P. Serpette
Arnaud Spiwack
Pawel Urzyczyn
Benjamin Wack
Marc Vesin

Un grand merci à mon Jury...
A massive thanks to my Jury...

Kim Bruce, Pomona, USA, Ref.
Gilles Dowek, LIX, France, Ref.
Robert Harper, CMU, USA, Ref.
Furio Honsell, UNIUD, IT, Memb.
Jacques Jaray, INPL, FR, Memb.
Claude Kirchner, INRIA, FR, Memb.
Pierre Lescanne, ENSL, FR, Memb
Michael Rusinowitch, INRIA, FR, Inv.
Horatiu Cirstea, NCYII, FR, Inv.

Un grand merci à ceux qui m'aiment et vivent dans les villes «touchées» par mon pèlerinage scientifique...
A massive thanks to all people that love me in all the "touched" towns in my scientific pilgrimage...

Pescara, Abruzzo, IT	**[A...Z]***
Udine, Friuli V.G., IT	**[A...Z]***
Torino, Piemonte, IT	**[A...Z]***
Lyon, Rhône-Alpes, FR	**[A...Z]***
Nancy, Lorraine, FR	**[A...Z]***
Sophia Antipolis, PACA, FR	**[A...Z]***



E-mail: Luigi.Liquori@sophia.inria.fr
Web: http://www-sop.inria.fr/mascotte/Luigi.Liquori/HDR/Peter.Pan

Contents

List of Tables

Comment lire cette thèse / How to Read this Thesis

C'est mon privilège et plaisir d'introduire Peter, *le langage qui n'existe pas...*

Le langage Peter contient quasiment tous les aspects linguistiques que j'ai introduits et étudiés dans le domaine de la programmation fonctionnelle et objets, ainsi que quelques idées qui n'ont pas encore été publiées.
Dans *l'habilitation de* Peter, la démarche que je suivrai consiste à essayer de limiter les détails concernant les aspects théoriques et techniques (c-à-d. les ensembles complets des règles de typage, suites de théorèmes abscons, etc.). Mon mémoire d'habilitation ne sera pas une traduction brutale des différents articles publiés[1], ni un curriculum vitæ commenté, ni un panorama de tous les articles dans un domaine scientifique[2], pour ne citer que quelques styles classiques de thèses d'habilitation. Tout d'abord elle sera courte car l'expérience enseigne que très peu de thèses d'habilitation sont réellement téléchargées, citées et lues. Très souvent, les thèses d'habilitation ne sont même pas accessibles sur le Web.

L'Habilitation de Peter sera fondée sur les trois «dogmes» suivants:

• *(Modularité)* Je commencerai par le plus petit fragment complet (au sens de Turing) de Peter, appelée Baby Peter et je continuerai de façon *modulaire*, d'extension en extension, jusqu'à l'extension finale appelée Sage Peter. Baby Peter est un langage fonctionnel avec des constructions linguistiques orientées objet et un système de types correct. Peter partage quelques similitudes avec *Featherweight Java* de Atsushi, Benjamin et Phil [IPW01] et le lambda calcul typé de Alonso (Church) [Chu41]. La différence prin-

It is my privilege and pleasure to introduce Peter, *the language that does not exist...*

The Peter language contains almost the linguistic features I have introduced and investigated in the field of functional and object-oriented programming, plus some new features not published yet.
In Peter*'s Habilitation*, I will try to limit as much as possible the mathematical overhead and the technicalities (*e.g.* full set of rules, full proofs of theorems, etc.). In my opinion, the habilitation thesis should not be a mere translation of the candidate's most successful papers[3], nor a commented curriculum vitæ, nor a survey of all the related works in his scientific area[4], just to mention a few "classic Habilitation styles". It is my opinion that it should be short in length since it is experienced that a very few Habilitation thesis are really downloaded, cited and read. Oftenly, habilitation thesis are not even made accessible on the Web.

Peter's Habilitation will be based on the following three points:

• *(Modularity)*) I will present a (Turing complete) kernel of Peter, called Baby Peter, and I will continue in the rest of the Habilitation to extend it in a modular fashion until the final extension, called Wise Peter. Baby Peter is a functional language with object-oriented features equipped with a sound type system. Peter bears some similarities to Atsushi, Benjamin and Phil's *Featherweight Java* [IPW01] and Alonso Church's typed lambda calculus [Chu41]. The main difference lies in an *ad hoc* exception-handling mechanism allowing the

cipale entre Featherweight Java et Peter, est un mécanisme d'exceptions *ad hoc*, qui permet au programmeur de décider quel système de types sera le plus adapté à l'egard de ses nécessités et objectifs. En plus, ce mécanisme permet au programmeur d'écrire son système de types (voir point *Type-programmable*). Certains chapitres seront focalisés sur un nouveau système de types, tandis que, dans d'autres chapitres, l'extension sera associée à une extension de la syntaxe *et* du système de types. Tous les arguments seront traités d'une façon accessible au plus grand nombre de lecteurs. Comme exemples d'extensions, je citerai une forme nouvelle d'héritage multiple, une extension de Peter qui permettra à un objet de «s'échapper de sa classe», une extension de Peter avec filtrage évolué et enfin une extension de Peter qui permettra de mélanger algorithmes *et* preuves de correction d'algorithmes.

• *(Verbatim-like)* Plutôt que d'asséner à mes lecteurs une traduction française mot-à-mot de mes articles scientifiques[5], j'ai privilegié une présentation simple de chaque extension, utilisant uniquement quelques règles clés de la sémantique opérationnelle ou du système de types (il y a toujours une règle clé...), en ajoutant immédiatement des exemples pour motiver et comprendre son utilisation correcte. Je ne prouverai pas la propriété de complétude de chaque système de types qui étend Peter : la complétude de Sage Peter est proposée en défi au prochain assistant à la preuve convivial.

• *(Type-programmable)* Les systèmes de types pour les langages de programmation et pour la preuve sont fixés *a priori* par leurs concepteurs et ne sont pas des *objets de première classe* pouvant être modifiés ou simplement utilisés par le programmeur qui en subit les qualités et les faiblesses. À ma connaissance, aucun langage ne permet au programmeur de «programmer» sa discipline de types personnelle. L'idée de modifier la discipline de typage à la compilation n'est pas très nouvelle ; un article «visionnaire» de 6 pages, qui m'a eclairé, a été *Pluggable Type System* de Gilad [Bra04] sorti en 2004. La possibilité de permettre au programmeur d'écrire sa propre discipline de typage et de l'utiliser à la volée est par elle-même une contribution originale dans *l'habilitation de* Peter.

programmer to choose the type system according to her/his necessities and goals. Even more, it allows the programmer to write her/his own type system (see item *(Type-programmable)*). Some chapters will focus on operational semantics, some others on type systems, some others on both. All topics will be treated in a "lightweight fashion". Examples of extensions are for instance mixing class-based and pure object-based features, but also improving proof languages *à la* LF with pattern matching facilities and including those metalanguages to Peter in order to mix algorithms *and* their correctness proofs.

• *(Verbatim-like)* Instead of annoying the reader with a plain French translation of some of my most relevant papers[6], I will show, for each extension, only some key rules of the operational semantics or of the type system (every system has at least a key rule...) and some motivating examples. I do not plan to prove type soundness for each extension of Peter: the whole soundness of Wise Peter is left as a challenge for the "next" user friendly proof assistant.

• *(Type-programmable)* Type systems for programming languages and proof languages are fixed *a priori* by language designers; type systems are not *first class citizens*. To my little knowledge, no language allows the programmer to build, choose, or mix type systems. The idea of modifying the type discipline at compile time is not completely new; a quite inspiring work has been done by the "visionary-6-pages" paper by Gilad in 2004 [Bra04] called *Pluggable Type Systems*. The possibility to mixing type systems and using it as a first class citizens is an interesting research strand that will constitute an original contribution in Peter*'s Habilitation*.

Avec l'envie de diffuser la connaissance scientifique de façon simple, claire et pédagogique, inspiré par les ouvrages de Kim [Bru99, TKB01, BDKT03, RBC+05, Bru02] et Gilles [Dow03, Dow07], il ne me reste plus qu'à vous souhaiter une bonne lecture de *l'habilitation de* Peter.

With the intention of disseminating science in a simple, clear and pedagogical way, and inspired by the works of Kim [Bru99, TKB01, BDKT03, RBC+05, Bru02] and Gilles [Dow03, Dow07], I wish you a very nice reading of the Peter*'s Habilitation.*

1 Bien que certaines parties soient tirées de mes articles.

2 La convention typographique est que les référence à mes articles soit en style «numérique» tandis que les références à d'autres articles soit en «alphanumérique».

3 Although certain parts are taken of my articles.

4 The typographic convention is that references to my papers are in "numeric" style while references to other papers are in "alphanumeric" style.

5 Un CD et un site web contiendront tous mes articles.

6 We provide a CD and a Web site with all my papers.

Sophia Antipolis, Été 2007

Luigi Liquori

Chargé de Recherche INRIA

Sophia Antipolis, Summer 2007

Luigi Liquori

INRIA Senior Researcher

Part I

Le jeune Peter (apprentissage) / The Young Peter (Learning)

Chapter 1

Peter apprend à apprendre / Peter Learns How to Learn

Ce chapitre est dédié à Gilad,
pour ses systèmes de types «pluggables» :
j'espère avoir poussé un peu plus loin
son idée dans cette thèse d'habilitation.

This chapter is dedicated to Gilad
for its pluggable type systems:
I hope to have pushed a bit more further
his ideas in this Habilitation Thesis.

1.1 Le petit Peter / Baby Peter

Les systèmes de types pour les langages de programmation et pour la preuve sont fixés *a priori* par leurs concepteurs ; en d'autres termes, les systèmes de types ne sont pas des *objets de première classe*. À ma connaissance il n'y a pas de langage de programmation qui permette au programmeur de choisir ou même de construire, son système de types. L'idée de modifier la discipline de type à la compilation est récente ; un article très intéressant est celui de Gilad en 2004 [Bra04] : dans cet article, Gilad préconise des langages où un système de types peut être «*collé*» à un langage sans que la sémantique opérationnelle du langage en soit affectée. Ma thèse essaye d'aller un petit peu plus loin que les idées visionnaires de Gilad. On voudrait définir, utiliser et manipuler plusieurs systèmes de types dans le même langage, comme des objets de première classe. Pour cela, on a inventé Peter le premier langage avec de vrais systèmes de types *collables*.

Type systems for programming languages and proof languages are fixed *a priori* by language designers; type systems are not *first class citizens*. To my knowledge, no language allows the programmer to build, choose or modify the type system. The idea of modifying the type discipline at compile time is quite recent; a quite inspiring work has been done by Gilad in 2004 [Bra04]. My thesis aims to go a little step further the visionary ideas of Gilad. My goal is to be able to define, use, and manipulate, as a first-class objects, a number of type systems inside the very same language. In order to do that, we devised Peter the first language with *pluggable* type systems.

On commence notre histoire avec Baby Peter, *i.e.* un langage fonctionnel et objets avec un système de types sûr. Baby Peter est inspiré par *Featherweight Java* de Atsushi, Benjamin

We begin our story with Baby Peter, *i.e.* a functional language with object-oriented features and a sound type system. Baby Peter is based on Atsushi, Benjamin and Phil's *Featherweight*

et Phil, mis à part d'une importante extension *ad hoc*, qui laissera au programmeur la possibilité de choisir son système de types et, même, de pouvoir définir son propre système. La syntaxe de Baby Peter est la suivante :

Java with a notable *ad hoc* extension that allows the programmers to decide which type system use and even to write their own type system. The syntax of Baby Peter is as follows:

CL	::=	class C extends D $\{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\}$	Class Declarations
K	::=	$\texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{f}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\}$	Constructors
M	::=	$\texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{return e};\}$	Methods
e	::=	x | e.f | e.m($\overline{\texttt{e}}$) | new C($\overline{\texttt{e}}$) | (C)e | usetype {e} in {e}	Expressions

Table 1.1: La syntaxe de Baby Peter / Baby Peter's Syntax

Dans l'expression usetype $\{e_1\}$ in $\{e_2\}$, e_1 est un système de types disponible dans la librairie du langage ou défini par le programmeur, tandis que e_2 est une expression qu'on cherchera à typer dans e_1. Un programmeur pourra, dans le même programme, utiliser différents systèmes de types disponibles pour Peter. L'expression usetype $\{e_1\}$ in $\{e_2\}$ permettera aux différents systèmes de types de se combiner selon une *visibilité lexicale* ; des règles de compatibilité (méta sous-typage) entre systèmes de types seront nécessaires pour renforcer la sûreté d'un programme.

In the expression usetype $\{e_1\}$ in $\{e_2\}$, e_1 is a type system which can be user defined or already available in the library of the language. e_2 is a program that will be type-checked using the type discipline defined in e_1. A programmer can use different type systems available for Peter. The expression usetype $\{e_1\}$ in $\{e_2\}$ will enforce *lexical scoping* between different type systems and some *compatibility* rules between type systems will be needed to enforce compatibility between type systems (a sort of "subtyping meta-systems").

1.1.1 Quelques règles à retenir / Some Rules to Retain

Les règles de typage suivantes sont très importantes pour comprendre le fonctionnement de la sémantique opérationnelle de Baby Peter :

The following simple rules are crucial in order to understand the correct behavior of the operational semantics of Baby Peter:

$$\frac{\texttt{CT(C)} = \texttt{class C extends D } \{\ldots\}}{\texttt{C <: D}}\ (\text{Sub·Class}) \qquad \frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e}_0)}{(\texttt{new C}(\overline{\texttt{e}})).\texttt{m}(\overline{\texttt{d}}) \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C}(\overline{\texttt{e}})/\texttt{this}]\texttt{e}_0}\ (\text{Run·Call})$$

$$\frac{\texttt{B m } (\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}} \quad \texttt{CT(C)} = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\ \overline{\texttt{TA}}\}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e})}\ (\text{MBody·Class})$$

$$\frac{\texttt{m} \notin \mathit{meth}(\overline{\texttt{M}}) \quad \texttt{CT(C)} = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\ \overline{\texttt{TA}}\}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{D})}\ (\text{MBody·SClass})$$

Table 1.2: Quelques règles d'exécution de Baby Peter / Baby Peter's Execution Rules

Dans une «coquille de noix»[1] ces règles montrent comment des programmes en Peter évoluent à l'exécution :
• (Sub·Class) Cette règle détermine le soustypage nominal ;
• (Run·Call) Cette règle traite l'appel d'une méthode : l'algorithme de liaison dynamique est calculé à l'aide de la fonction *mbody* ; elle est utilisée également dans les règles de coercion ;
• (MBody·Class) Cette règle traite la liaison dynamique : la méthode est trouvée dans la classe courante ;
• (MBody·SClass) Cette règle traite aussi la liaison dynamique : la méthode est trouvée dans une super-classe directe.

In a nutshell, these rules dictate how Peter's programs evolve at run-time.
• (Sub·Class) This rule is the usual nominal subtyping;
• (Run·Call) This rule deals with method call: dynamic method lookup via the auxiliary function *mbody*; it is also used in the coercion runtime rules;
• (MBody·Class) This rule deals with method lookup: the method is found in the current class we are looking for;
• (MBody·SClass) This rule deals with method lookup: the method is found in one of its direct inherited superclass.

1 Les traducteurs automatiques sont très surprenants : j'ai laissé volontairement la traduction de *"In a nutshell"* par «*dans une coquille de noix*», au lieu de la vraie traduction «*en quelques mots*».

$$\frac{\texttt{CT(C)} = \texttt{class C extends D } \{\ldots\} \quad \texttt{F <: E} \quad \overline{\texttt{x}}{:}\overline{\texttt{C}}, \texttt{this:C} \vdash \texttt{e} : \texttt{F} \quad \mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{E})}{\texttt{E m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{return e;}\}\ \texttt{OK IN C}}\ (\text{Meth·Ok·Class})$$

$$\frac{\texttt{K} = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{g}})\texttt{; this.}\overline{\texttt{f}} = \overline{\texttt{f}}\texttt{;}\} \quad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \quad \overline{\texttt{M}}\ \texttt{OK IN C}}{\texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \texttt{ OK}}\ (\text{Class·Ok})$$

Table 1.3: Règles clés pour typer une méthode/classe en Baby Peter / Baby Peter's Key Typing Rules for Methods/Classes

Ces deux règles, directement héritées de Featherweight Java, sont la clé du fonctionnement du système de types.

These two rules, directly inherited by Featherweight Java, dictate the correct behavior of the type-checking system.

Dans une «coquille de noix» :
• (Meth·Ok·Class) Cette règle vérifie si une méthode `m` est bien typée dans la classe `C` : on type le corps de la méthode dans un contexte de types enrichi des paramètres actuels et de la méta-variable `this`[1]. Le type du corps doit être un sous-type du type déclaré pour le résultat. Le prédicat *override* vérifie la conformité du type par rapport à l'héritage, c-à-d. si la méthode est héritée, alors les types des paramètres formels et du résultat ne changent pas (invariance) ;
• (Class·Ok) Cette règle type une classe *in toto*. Toutes les méthodes contenues ou héritées dans la classe doivent être bien typées, tandis que les variables définies et héritées doivent être déclarés dans la super-classe directe.

In a nutshell:
• (Meth·Ok·Class) This rule checks whether a method `m` is well-typed in the class `C`: it first type-checks the body of `m` in `C` in a context enriched with the types of the actual parameters and the type of the metavariable `this`[2]. Then it checks whether the resulting type for the body is a subtype of the declared return type; finally, thanks to the *override* predicate, it ensures the absence of any overloading of an inherited method or of any illegal overriding, *i.e.*, overloadings changing the type of an already defined method with the type of parameters and of the result;
• (Class·Ok) This rule deals with the typing of the class itself. All methods contained in, or inherited by the class must be well-typed; all inherited fields must be declared in the superclass.

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}} \text{ (Type·Call)}$$

Table 1.4: La règle de typage qui appelle une méthode en Baby Peter's / Baby Peter's Type Rule for Method Call

La règle (Type·Call) ne contient pas de surprises : on type l'objet qui reçoit le message et on calcule le type de la méthode que l'on doit exécuter ; on vérifie la compatibilité du type des arguments et on rend le type du résultat de la méthode.

The (Type·Call) rule is no surprise at all: we typecheck the receiver, and we fetch the functionality of the method we are calling; then we check the compatibility of the arguments and we give as result the type of the call.

1.1.2 Installe ton système de types / Plug your Type System

La partie nouvelle de Baby Peter est que la discipline de typage n'est pas fixée *a priori*, mais elle peut être changée par le programmeur, en accord avec des règles de «méta-compatibilité» entre les système de types utilisés. Un système de types peut faire partie de la librairie du langage ou peut être défini directement par le programmeur ; cela fait des systèmes de types des objets de première classe. Les règles de la sémantique opérationnelle et du système de types pour ce nouveau constructeur de Peter sont :

The novel part in Baby Peter is that the type system discipline is not fixed *a priori* but can be changed by the programmer, according to some "meta-compatibility" rules among different type systems. A type system can be part of the language library, or it can be defined by the user. As such, in Baby Peter, type systems are actually fist-class citizens. The run-time semantics and the static type checking rules of this new Peter constructor are:

[2]Si on élimine `this` du contexte, on obtient Norm Peter, un langage sans récursion.
[3]By dropping `this` from the context, we obtain Norm Peter, a language without recursion.

$$\frac{}{\texttt{usetype}\ \{\texttt{e}_1\}\ \texttt{in}\ \{\texttt{e}_2\} \longrightarrow \texttt{e}_2}\ (\text{Run·Ok·Type}) \qquad \frac{\Gamma \vdash_{\texttt{e}} \texttt{e}_1 : \texttt{C} \quad \vdash_{\texttt{e}_1} \sqsubseteq \vdash_{\texttt{e}} \quad \Gamma \vdash_{\texttt{e}_1} \texttt{e}_2 : \texttt{D}}{\Gamma \vdash_{\texttt{e}} \texttt{usetype}\ \{\texttt{e}_1\}\ \texttt{in}\ \{\texttt{e}_2\} : \texttt{D}}\ (\text{Type·Typecheck})$$

Table 1.5: Les règles pour exécuter et typer une expression qui introduit un système de types / Run-time and Type Rules for Pluggable Type Expressions

Dans une «coquille de noix» :
• (Run·Ok·Type) Cette règle dérobe l'expression $\texttt{e}_2$ et l'exécute sans aucun test de type dynamique ;
• (Type·Typecheck) Cette règle *change la discipline de typage* ; tout d'abord le nouveau système de types $\texttt{e}_1$ doit être typable dans le système de types dans lequel l'expression $\texttt{usetype}\ \{\texttt{e}_1\}\ \texttt{in}\ \{\texttt{e}_2\}$ se trouve statiquement (par exemple le système $\vdash_{\texttt{e}}$) ; on vérifie ensuite que $\texttt{e}_1$ est un véritable système de types, c-à-d. une fonction qui prend une expression de Peter et produit un booléen (typable ou non) : $\vdash_{\texttt{e}_1}$ doit être compatible avec le système de types précédent qui est en train d'être «assombri» par le nouveau ; finalement, on essaye de typer l'expression $\texttt{e}_2$ dans le nouveau système $\vdash_{\texttt{e}_1}$. La relation $\vdash_{\texttt{e}_1} \sqsubseteq \vdash_{\texttt{e}}$ pourra être formalisée comme suit :

In a nutshell:
• (Run·Ok·Type) This rule just runs the program $\texttt{e}_2$ without any dynamic check;
• (Type·Typecheck) This rule *changes the type system discipline*; first of all one must check that $\texttt{e}_1$ typechecks in the original type system (say system $\vdash_{\texttt{e}}$), where the expression $\texttt{usetype}\ \{\texttt{e}_1\}\ \texttt{in}\ \{\texttt{e}_2\}$ occurs statically; second, one must check that $\texttt{e}_1$ is a type checker, *i.e.* a function from expressions to booleans (typable or not); this system $\vdash_{\texttt{e}_1}$ must be compatible with the previous type system that is going to be "obfuscated" by the new one; finally, one must type the expression $\texttt{e}_2$ against the new type system $\vdash_{\texttt{e}_1}$. The relation $\vdash_{\texttt{e}_1} \sqsubseteq \vdash_{\texttt{e}}$ can be formalized as follows:

Definition 1 (Compatibilité entre systèmes de types / Type Systems Compatibility)

Pour chaque $\vdash_{\texttt{e}_1}, \vdash_{\texttt{e}_2} \in$ [Expr. $\to$ Bool], *on dit que* $\vdash_{\texttt{e}_1} \sqsubseteq \vdash_{\texttt{e}_2}$ *si et seulement si* $\vdash_{\texttt{e}_1}$ *préserve au moins les mêmes «*erreurs capturées*» [Car04] par* $\vdash_{\texttt{e}_2}$, *sur les expressions analysées par* $\vdash_{\texttt{e}_1}$ *et* $\vdash_{\texttt{e}_2}$ *ou sur les expressions analysées par* $\vdash_{\texttt{e}_1}$ *et non par* $\vdash_{\texttt{e}_2}$. *Un peu plus formellement, soit* $\mathcal{L}(\vdash)$ *l'ensemble des expressions du langage* $\mathcal{L}$ *analysées par* $\vdash$ *:*

For every $\vdash_{\texttt{e}_1}, \vdash_{\texttt{e}_2} \in$ [Expr. $\to$ Bool], *we say that* $\vdash_{\texttt{e}_1} \sqsubseteq \vdash_{\texttt{e}_2}$ *if and only if* $\vdash_{\texttt{e}_1}$ *preserves at least the same "*catched errors*" [Car04] of* $\vdash_{\texttt{e}_2}$, *on expressions parsed by both* $\vdash_{\texttt{e}_1}$ *and* $\vdash_{\texttt{e}_2}$ *or on expressions parsed by* $\vdash_{\texttt{e}_1}$ *but not by* $\vdash_{\texttt{e}_2}$. *A bit more formally, let* $\mathcal{L}(\vdash)$ *be the set of expressions of* $\mathcal{L}$ *parsed by* $\vdash$*:*

$$\vdash_{\texttt{e}_1} \sqsubseteq \vdash_{\texttt{e}_2} \iff \Gamma \vdash_{\texttt{e}_1} \texttt{e} : \sigma \Longrightarrow \begin{cases} (either) & \Gamma \vdash_{\texttt{e}_2} \texttt{e} : \sigma \quad \forall \Gamma, \texttt{e} \in \mathcal{L}(\vdash_{\texttt{e}_1}) \cap \mathcal{L}(\vdash_{\texttt{e}_2}) \\ (or) & \Gamma \not\vdash_{\texttt{e}_2} \texttt{e} \qquad \forall \Gamma, \texttt{e} \in \neg(\mathcal{L}(\vdash_{\texttt{e}_1}) \cap \mathcal{L}(\vdash_{\texttt{e}_2})) \end{cases}$$

Dans une «coquille de noix», le (*either*)*-cas dit qu'un système de types plus «rigide» peut être utilisé dans tout contexte qui attend un système de types plus «souple», à condition que les propriétés capturées par le système plus souple soient préservées. Le* (*or*)*-cas dit que si on étend le langage, alors une expression strictement propre à l'extension sera typable dans le*

In a nutshell, the (*either*)*-case says that a "more strict" type system can be used in any context expecting a "less strict" type system, proviso the catched errors by the weaker system being preserved. The* (*or*)*-case says that if we extend the language, then the expression belonging to the extension will be typable in the "pluggable" type system proviso it will preserve the*

système qu'on vient «d'attacher», à condition qu'il préserve les mêmes propriétés du langage qu'on étend. À noter que la rélation $\sqsubseteq$ n'est pas transitive.

same catched errors of the extended language. Note that the relation $\sqsubseteq$ is not transitive.

Dans l'Annexe A on résumera les compatibilités entre les systèmes de types de Peter.

In Appendix A, all Peter's type systems compatibilities will be resumed.

1.1.3 Un exemple motivant / A Motivating Example

Par exemple, considérons deux systèmes de types, le premier appelé `normalizing` et le deuxième appelé `recursive`, tels que `normalizing` $\sqsubseteq$ `recursive`. Une `expression` (sans entrée ni sortie) qui sera typable avec `normalizing` terminera sûrement à l'exécution, tandis qu'une expression qui sera typable avec `recursive` pourra implanter la récursion et, donc, *a priori*, ne pas terminer. La relation de tolérance pourrait affirmer que, si une expression est compilée avec le système `normalizing`, alors elle pourrait être utilisée dans une autre expression qui sera compilée avec le système `recursive` (l'affirmation réciproque n'étant pas vraie). Par exemple :

For example, consider two type-system expressions, the first one called `normalizing`, and the one second called `recursive` such that `normalizing` $\sqsubseteq$ `recursive`. An expression (without I/O) typable with `normalizing` will surely terminate, while an expression typable with `recursive` can loop forever. The "tolerance" relation should enforce that if an expression is typechecked with `normalizing`, then it can be used inside another expression typechecked with `recursive` (the vice-versa being obviously not true). For example:

```
usetype {recursive} in {expression_1
                        ...
                        usetype {normalizing} in {expression_k}
                        ...
                        expression_n}
```

Table 1.6: Utilisation correcte de `usetype` / Correct use of `usetype`

se compile si `expression_1,2...k-1,k+1...n` compilent dans `recursive` et si `expression_k` se compile dans `normalize`. En fait, on peut tolérer la routine `expression_k` (qui termine) dans un programme récursif entier qui potentiellement pourrait ne jamais terminer. Par contre, la relation de tolérance ne pourra pas accepter la réciproque, ainsi le programme suivant ne sera pas typable :

typechecks if `expression_1,2...k-1,k+1...n` typechecks with `recursive` and `expression_k` typechecks with `normalize`. Thus, we can accept the terminating routine `expression_k` inside a recursive program which can potentially loop forever. The reverse should be not accepted by the tolerance rules, *i.e.* the following program would be untypable:

```
usetype {normalizing} in {expression_1
                          ...
                          usetype {recursive} in {expression_k}
                          ...
                          expression_n}
```

Table 1.7: Utilisation incorrecte de `usetype` / Incorrect use of `usetype`

car on ne pourra jamais accepter une routine récursive dans un programme qui terminera grâce à une discipline de typage très rigide.

since we cannot accept a potentially non-terminating recursive routine inside a program that must be guaranteed to terminate.

Sources d'inspiration / Inspiration sources

Ce chapitre a été inspiré par l'article Featherweight Javade Atsushi, Benjamin et Phil [IPW01] et par l'article de Gilad [Bra04] sur les systèmes de types «collables». Le lecteur intéressé par les langages à classes et leurs théories des types trouvera son bonheur dans les excellents ouvrages de Benjamin [Pie02, Pie05].

This chapter has been influenced by Atsushi, Benjamin, and Phil's Featherweight Java [IPW01] and by the inspiring work of Gilad [Bra04] on pluggable type systems. The interested reader in typed class-based language can have a look to the excellent Benjamin's books [Pie02, Pie05].

Chapter 2

Peter apprend avec deux mains / Peter Learns with Two Hands

Ce chapitre est dédié à Mariangiola et Simona : «merci» de m'avoir appris à «deux^{deux} mains» ce métier excitant.

This chapter is dedicated to Mariangiola and Simona: "thanks" of having taught me, with "two^{two} hands", this exciting job.

2.1 Héritage par les traits / Inheritance via Traits

2.1.1 Syntaxe / Syntax

«Dans chaque gros langage il y a un petit langage qui lutte pour en sortir...»

"Inside every large language is a small language struggling to get out..." [IPW01]

«...et dans chaque petit langage il y a une extension astucieuse qui veut plus d'expressivité...»

"...and inside every small language is a sharp extension looking for better expressivity..."

Dans le contexte des langages typés statiquement et basés sur les classes, nous avons étudié un mécanisme pour étendre les classes par la composition de *traits*. Intuitivement, un trait est une collection de méthodes sans état. Un trait est donc une classe incomplète sans variables. Les traits peuvent être combinés dans n'importe quel ordre. Un trait est utilisable seulement quand il est importé dans une classe qui possède des variables et des méthodes pour désambiguïser les conflits de noms entre les traits importés.

In the context of *statically-typed, class-based languages*, we investigate classes that can be extended with *trait* composition. Intuitively, a trait is just a collection of methods, *i.e.* behaviors without state. Thus, traits are (potentially incomplete) classes without state. Traits can be composed in any order. A trait only makes sense when "imported" by a class that provides state variables and possibly some additional methods to disambiguate conflicting names arising between the imported traits.

Ce fil de recherche nous a menés à ajouter dans Baby Peter un mécanisme de traits typés. On

This research strand led us to add *statically-typed traits* to Baby Peter. We call this exten-

appelle cette extension Jeune Peter. Les conflits entre les méthodes doivent être résolus explicitement par l'utilisateur soit (1) en renommant ou en excluant la méthode, soit (2) en redéfinissant explicitement les méthodes concernées dans la classe. Comme dans tous les problèmes liés à l'héritage multiple, le problème du *diamant* peut apparaître.
Nous avons présenté une nouvelle sémantique opérationnelle avec un nouvel algorithme de recherche de méthode et un nouveau système de types qui garantit que l'évaluation ne soulève pas l'erreur «message-non-compris» et que l'interprète ne se bloque pas. Le calcul proposé semble être un bon point de départ pour une analyse mathématique rigoureuse des langages typés basés sur les traits.

sion Young Peter. Method conflicts between imported traits must be resolved *explicitly* by the user either by (1) aliasing or excluding method names in traits, or by (2) overriding explicitly the conflicting methods in the class or in the trait itself. We emphasize "diamond conflicts", a classical issue in trait inheritance.

We present an operational semantics with a lookup algorithm, and a sound type system which guarantees that evaluating a well-typed expression never yields a *message-not-understood* run-time error nor gets the interpreter stuck. We give simple examples which illustrate the increased expressive power of the trait-based inheritance model of Young Peter with respect to the classical single inheritance model. The resulting calculus appears to be a good starting point for a rigorous mathematical analysis of typed class-based languages featuring trait inheritance.

L'algorithme de liaison dynamique implante les trois règles suivantes :
• Une méthode définie dans une classe est prioritaire sur les méthodes définies dans les traits importés dans la classe ;
• Une méthode définie dans un trait est prioritaire sur les méthodes importées dans le trait ;
• Une méthode définie dans un trait et importée par une classe est prioritaire sur la méthode définie dans la super-classe.

Three simple rules can be easily implemented in the method-lookup algorithm for that purpose
• Methods defined in a class take precedence over methods defined in the traits imported by the class;
• Methods defined in a composite trait take precedence over methods defined in the imported traits;
• Methods defined in traits (imported by a class) take precedence over methods defined in its parent class.

CL	::=	class C extends D [imports $\overline{\texttt{TA}}$] {$\overline{\texttt{C}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$}	Class Declarations
TL	::=	trait T [imports $\overline{\texttt{TA}}$] {$\overline{\texttt{M}}$}	Trait Declarations
TA	::=	T | TA with {m@n} | TA minus {m}	Trait Alterations

Table 2.1: Syntaxe de Jeune Peter avec des traits / Young Peter's Syntax with Traits

2.1.2 Exemples / Examples

Une autre propriété de l'héritage par trait est qu'une classe qui importe un trait est équivalente à une classe qui définit *in situ* toutes les méthodes qui sont définies dans le trait importé. Ceci est possible grâce à une technique dite d'*aplatissement*, qui, tout de

Another property of trait-based inheritance is that a class that imports traits is semantically equivalent to a class that defines *in situ* all the methods defined in traits. This can be done via *flattening*, which immediately suggests how to build a compiler translating Young Peter code

suite, nous suggère comment implanter un compilateur qui traduit du code de Jeune Peter avec des traits dans du code pur de Baby Peter, via une duplication de code.

with traits into pure Baby Peter code, via code duplication.

On commence, ici, par définir trois traits :

We start to define three traits:

```
trait T1 {String p(){return ‘‘hello’’;}}
trait T2 {String p(){return ‘‘world’’;}}
trait T3 imports T1,T2
        {String m(){return (...this.p()...);}          p is a required method
         String p(){return ‘‘hello world’’;}        p is an overridden method
         String n(){return (...this.q()...);}          q is a required method
        }     Trait T3 imports traits T1 and T2, override p, and q is still a required method
```

Table 2.2: Trois traits simples / Three Simple Traits

On observe que dans le trait `T3`, qui importe les traits `T1` et `T2`, la méthode `p` est redéfinie et la méthode `q` invoquée par `n` est absente. Donc un trait est, par définition, potentiellement *incomplet*, c-à-d., qu'il ne peut pas être instancié dans un objet «vivant», car il ne peut pas avoir d'état et l'implantation de certaines méthodes peut manquer.

Observe that in trait `T3`, that imports traits `T1` and `T2`, the method `p` is redefined and the method `q`, invoked by `n` is absent. As such, a trait is, by definition, *possibly incomplete*, *i.e.*, it cannot be instantiated in a "runnable" object, since it cannot have any state, and it can lack some method implementations.

```
trait T4 {Object p(){return(...this.r()...);}          r is a required method
trait T5 {Object q(){return(...this.s()...);}          s is a required method
trait T6 imports T4,T5
        {Object m(){return(...this.p()...);}           p is a required method
         Object n(){return(...this.q()...);}           q is a required method
        }          Trait T6 imports traits T4 and T5 and r and s are still required methods
```

Table 2.3: Encore trois traits simples / Still Three Simple Traits

Dans la composition des traits, des conflits peuvent se produire : par exemple, une classe `C` peut importer deux traits `T1` et `T2` qui définissent deux corps différents pour la méthode `p`. Les conflits entre les traits doivent être résolus *à la main*, c-à-d. qu'il n'existe pas de règle spéciale et rigide pour apprendre à utiliser les traits typés. Il existe trois façons de résoudre les conflits entre des traits (dans la suite, le «gagnant» représente le corps qui est sélectionné par l'algorithme de liaison dynamique) :

When dealing with trait inheritance, conflicts can arise; for example a class `C` might import two traits `T1` and `T2` defining the same method `p` with different behaviors. Conflicts between traits must be resolved *manually*, *i.e.* there is no special or rigid discipline to learn how to use traits. Once a conflict is detected, there are essentially three ways to resolve the conflict (below, "winner" denotes the body selected by the lookup algorithm):

Redéfinir une nouvelle méthode `p` dans la classe. Une nouvelle méthode `p` est redéfinie dans la classe avec un nouveau comportement. L'algorithme de liaison dynamique (étendu aux traits) cachera le conflit entre le trait `T1` et `T2` en faveur de la nouvelle méthode définie dans la classe. Dans la syntaxe de Jeune Peter :

Overriding a new method `p` inside the class. A new method `p` is redefined inside the class with a new behavior. The (trait-based) lookup algorithm will hide the conflict in traits in favor of the overridden method defined in the class. In Young Peter syntax:

```
class C extends Object
        imports T1,T2                          each trait defines a (different) behavior for p
{...;...                                                          instance vars and constructor
 D p(...){...};}                                          new behavior for p, the winner
```

Table 2.4: Redéfinir `p` dans `C` / Overriding `p` inside `C`

Renommer la méthode `p` dans les traits et redéfinir une nouvelle méthode dans la classe. La méthode `p` est renommée dans `T1` et `T2` avec un nom différent. Un nouveau comportement pour `p` sera décrit dans la classe `C` (on pourra même réutiliser les méthodes des traits, renommées en `p_of_T1` et `p_of_T2`, qui ne seront plus en conflit). Dans la syntaxe de Jeune Peter :

Aliasing the method `p` in traits and redefining the method in class. The method `p` is aliased in `T1` and `T2` with new different names. A new behavior for `p` can be now given in the class `C` (possibly re-using the aliased methods `p_of_T1` and `p_of_T2` which are no longer in conflict). In Young Peter syntax:

```
class C extends Object
        imports T1 with {p@p_of_T1}[1],                    T1 aliases p with p_of_T1
                T2 with {p@p_of_T2}                        T2 aliases p with p_of_T2
{...;...                                                          instance vars and constructor
 D p(...){...}}                          new winner behavior for p, it may use p_of_T1/2
```

Table 2.5: Renommer `p` dans les traits et redéfinir `p` dans `C` / Aliasing `p` in Traits and Overriding `p` in `C`

Exclure la méthode `p` d'un trait. Une méthode `p` dans le trait `T1` ou `T2` est exclue. Cela résout le conflit en faveur d'un des deux traits. Dans la syntaxe de Jeune Peter :

Excluding the method `p` in one of the traits. One method `p` in trait `T1` or `T2` is excluded. This solves the conflict in favor of one trait. In Young Peter syntax:

```
class C extends Object
        imports T1,                                    contains the winner method p
                T2 minus {p}                                 method p is now hidden
{...;...}                                                     instance vars and constructor
```

Table 2.6: Exclure `p` dans un trait / Excluding `p` in one Trait

Le tableau suivant introduit trois traits et quatre classes qui importent ces traits (certain étant renommés). Le tableau résume toutes les possibilités d'appel de méthodes (on suppose l'existence du type `int` et d'une algèbre pour les entiers) :

The following table defines three simple traits and four classes that import those traits (some of them aliased). The table summarizes all possible method calls (we assume types and algebras for integers).

```
T1 {int m(){return this.f+1;}
    int n(){return this.f*10;}
    int p(){return this.q()+10;}}
T2 {int m(){return this.f+2;}
    int n(){return this.f*20;}
    int q(){return this.m()+this.n();}}
T3 {int m(){return this.f+3;}
    int n(){return this.f*30;}}
class A extends Object imports T1 minus {m},T2 minus {n}
      {int f; A(int f){super();this.f=f;}}
class B extends Object imports T1 minus {n},T2 minus {m}
      {int f; B(int f){super();this.f=f;}}
class C extends Object imports T1 with {m@m_T1},T2 with {n@n_T2}}
      {int f; C(int f){super();this.f=f;}
       int m(){return this.m_T1()+this.n_T2()}
       int n(){return this.m_T1()*this.n_T2()}}
class D extends A imports T3
      {; C(int f) {super(f);}
       int p(){return this.q()+100;}
       int q(){return this.m()*this.n();}}
```

receiver	this.m()	this.n()	this.p()	this.q()
new(A(3))	5	30	45	35
new(B(3))	4	60	74	64
new(C(3))	64	240	314	304
new(D(3))	6	90	640	540

Table 2.7: Devinez les résultats en $\leq$ 2 minutes / Guess Results in $\leq$ 2 Minutes

2.1.3 Un exemple amusant / A Funny Example

L'exemple suivant montre que l'utilisation des traits ne produit pas du code absurde. La possibilité de composer des traits qui contiennent les mêmes noms de méthodes avec des types différents est un des résultats clés de cette extension de Jeune Peter. La possibilité supplémentaire de détecter et de typer des méthodes héritées via l'héritage multiple des traits («diamant») est un autre avantage de Jeune Peter. En gros, cela correspond à accepter toutes les méthodes «sûres» *à la* Smalltalk avec traits [DNS+06] qui ne soulèveront pas

This example shows how traits do not break legal code. The ability to compose traits containing the same method name and different, incompatible, signatures is one of the key result of the present work. The ability also to detect and type-check innocuous methods inherited via multiple trait inheritance (a.k.a. via a diamond inheritance) is another achievement of Young Peter. Roughly speaking, this corresponds to accept all "safe" Smalltalk-like trait-based feature [DNS+06] that would not raise exceptions of the kind `message-not-understood` at run-

une exception du genre `message-pas-connu` à l'exécution.

time.

```
trait Freedom    {Independence declaration(){return ...;}
                  Human_Rights  acclamation(){return ...;}
                  Food         freedom_food(){return ``Turkey'';}
trait Democratic imports Freedom
                 {Kerry program(){return ...;}}
trait Republican imports Freedom
                 {Bush      program(){return ...;}
                  Food freedom_food(){return ``Freedom_fries'';}}
trait Outsider   imports Democratic with {program@program_demo}
                 {Chirac    program(){return ...;}
                  Food freedom_food(){return ``French_fries'';}}

class One_Candidate extends Object
                    imports Democratic
{...;
 One_Candidate(){super();}
 Object merge(){return ...
                this.declaration() ...
                this.acclamation() ...
                this.program() ...
                this.freedom_food() ...
                this.program_demo();}
 Object  ask(){return this.merge();}}

class Two_Candidate extends Object
                    imports Republicain
{...;
  Two_Candidate(){super();}
  Object  merge(){return ...
                  this.declaration() ...
                  this.acclamation() ...
                  this.program() ...
                  this.freedom_food() ...
                  this.program_repub();}
  Object    ask(){return this.merge();}}

class Three_Candidate extends Object
                      imports Outsider

 {...;
 Three_Candidate(){super();}
 Object merge(){return ...
                this.declaration() ...
                this.acclamation() ...
                this.program() ...
                this.program_demo();}
 Object   ask(){return this.merge();}}

class Four_Candidate extends Object
                     imports
 Democratic with {program@program_demo},
 Republican with {program@program_repub}
 {...;
 Four_Candidate(){super();}
 Object  merge(){return ...
                 this.declaration() ...
                 this.acclamation() ...
                 this.program() ...
                 this.program_demo();
                 this.program_repub();}
 Object     ask(){return this.merge();}
 Blair  program(){return ...;}
 Food freedom_food(){return
                    ``Fish_and_Chips'';}}

(new   One_Candidate()).ask()          a merge of Freedom, Kerry program, and Turkey
(new   Two_Candidate()).ask();      a merge of Freedom, Bush program, and Freedom fries
(new Three_Candidate()).ask()a merge of Freedom, Kerry and Chirac programs,and French fries
(new  Four_Candidate()).ask();a ''strange'' merge of Freedom, Blair, Bush,and Kerry programs
                                                              and Fish and Chips...
```

Table 2.8: Un exemple fort amusant en Jeune Peter[2] / A Funny Example in Young Peter[3]

2.1.4 Règles clés pour l'exécution / Key Run-time Rules

Les règles clés à retenir sont celles qui modifient l'algorithme de liaison dynamique.

The main rules to keep in mind are the ones that dictate the dynamic lookup algorithm.

$$\frac{\texttt{CT(C)} = \texttt{class C extends D imports } \overline{\texttt{TA}} \ \{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\} \quad \texttt{m} \notin \mathit{meth}(\overline{\texttt{M}}) \quad \mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e})}\ (\text{MBody·Trait})$$

$$\frac{\exists \texttt{TA} \in \overline{\texttt{TA}}.\ \mathit{altlook}(\texttt{m}, \texttt{TA}) \neq \mathsf{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \mathit{altlook}(\texttt{m}, \texttt{TA})}\ (\text{Trait·Ok}) \qquad \frac{\forall \texttt{TA} \in \overline{\texttt{TA}}.\ \mathit{altlook}(\texttt{m}, \texttt{TA}) = \mathsf{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \mathsf{fail}}\ (\text{Trait·Ko})$$

$$\frac{\texttt{TT(T)} = \texttt{trait T imports } \overline{\texttt{TA}}\ \{\overline{\texttt{M}}\} \quad \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{\mathit{altlook}(\texttt{m}, \texttt{T}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}\ (\text{ATrait·Found})$$

Table 2.9: Règles clés pour la liaison dynamique en Jeune Peter / Young Peter's Key Run-time Rules for Dynamic Lookup Algorithm

Dans une «coquille de noix» :
• (MBody·Trait) Cette règle cherche la méthode `m` ans tous les traits importés ;
• (Trait·∗) La fonction *tlook* ne recherche la méthode `m` que si `m` n'est pas redéfinie dans la classe ; cela force l'unicité de la recherche, autrement un conflit pourrait avoir lieu (car une méthode définie dans une classe pourrait redéfinir la méthode `m`). La fonction cherche le corps de la méthode `m` en traversant une suite de traits altérés ;
• (ATrait·Found) La fonction auxiliaire *altlook* prend en compte les traits altérés (c-à-d., les traits avec des méthodes exclues ou avec des méthodes renommées). Chercher une méthode qui a été exclue ou renommée est une des parties clés de l'algorithme de liaison dynamique. Dans cette règle, *altlook* rend un résultat qui est le corps de la méthode que l'on est en train de chercher, car cela signifie que la méthode est trouvée dans le trait.

In a nutshell:
• (MBody·Trait) This rule searches the method body of `m` in all imported traits;
• (Trait·∗) The trait lookup function *tlook* searches the method body of `m` only if `m` is not overridden in the class; this forces the uniqueness of the search, otherwise a conflict would arise (since a method defined in the class could have overridden method `m`). The function searches the body of the method `m` "traversing" a sequence of trait alterations;
• (ATrait·Found) The auxiliary function *altlook* takes into account altered traits (*i.e.*, traits with dropped methods, or with aliased methods). Finding a method which has been dropped or aliased is one of the key parts of the lookup algorithm. In this rule, *altlook* succeeds returning the body of the method we are looking for, since that method is found in the trait.

2.1.5 Règles clés pour le typage / Key Type-checking Rules

La règle pour typer un trait est la suivante :

The rule dealing with typing of trait is:

[2]Les noms utilisés n'ont été choisis que pour montrer la puissance de combinaison des traits avec des types différents. Il n'y a absolument pas de message politique ; d'ailleurs le lecteur francophone pourrait s'amuser à remplacer Bush par Sarkozy, Kerry par Royal et Chirac par Bayrou.

[3]Disclaimer: The names used here are chosen only for the purpose of showing the power of combining traits with different types. There is absolutely no political message inside.

$$\frac{\texttt{TT}(\texttt{T}) = \texttt{trait T imports } \overline{\texttt{TA}}\ \{\overline{\texttt{M}}\} \quad \overline{\texttt{N}} \triangleq \{\texttt{M} \in \overline{\texttt{M}} \mid \neg \texttt{M OK IN C}\} \quad \overline{\texttt{TA}} \texttt{ OK IN C except } \overline{\texttt{m}} \quad \cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}})}{\texttt{T OK IN C except } \mathit{meth}(\overline{\texttt{N}}) \cup (\overline{\texttt{m}} \setminus \mathit{meth}(\overline{\texttt{M}}))}\ (\text{Trait·Ok})$$

Table 2.10: La règle de typage pour T dans C en Jeune Peter/ Young Peter's Type Rule for T in C

Dans une «coquille de noix» :
• Le jugement T OK IN C except $\overline{\texttt{n}}$ dit qu'un trait TA est bien typé dans une classe C où les méthodes $\overline{\texttt{n}}$ doivent être redéfinies à nouveau. Le principe de cette règle est le suivant : chaque méthode qui apparaît dans la partie except concerne un corps de méthode qui ne peut pas être typé dans C et qui sera sûrement redéfini ailleurs. La règle vérifie aussi que chaque $\texttt{TA} \in \overline{\texttt{TA}}$ est bien typé. En quelques mots :
• On accède à la définition du trait T dans la table TT ;
• On détecte toutes les méthodes $\overline{\texttt{N}}$ définies dans le trait T qui ne sont pas typables dans C ;
• On type l'ensemble de traits altérés $\overline{\texttt{TA}}$ dans C en produisant un ensemble de méthodes *illégales* (qu'on prend dans la partie except $\overline{\texttt{m}}$) qui représentent les méthodes de $\overline{\texttt{TA}}$ qui ne typent pas dans C ;
• On vérifie la condition $\cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}})$ qui, intuitivement, garantit qu'il n'y a pas de conflit et que l'algorithme de liaison dynamique est déterministe ;
• On construit un nouvel ensemble de méthodes illégales $\mathit{meth}(\overline{\texttt{N}}) \cup (\overline{\texttt{m}} \setminus \mathit{meth}(\overline{\texttt{M}}))$, c-à-d. les méthodes illégales de TA $(\mathit{meth}(\overline{\texttt{N}}))$ plus les méthodes illégales de $\overline{\texttt{TA}}$ $(\overline{\texttt{m}} \setminus \mathit{meth}(\overline{\texttt{M}}))$ qui ne sont pas encore redéfinies, car les méthodes redéfinies $\overline{\texttt{M}}$ ne sont plus illégales.

In a nutshell:
• The judgment T OK IN C except $\overline{\texttt{m}}$ says that a trait TA is well-typed w.r.t. a given class C where all methods $\overline{\texttt{m}}$ must be overridden. The rationale of the rule is as follows: every method occurring in the except part refers to a body that cannot be type-checked in C, and it must be overridden by another. The rule also ensures that every $\texttt{TA} \in \overline{\texttt{TA}}$ is well-typed. In a nutshell:
• We fetch the trait T in the trait-table TT;
• We fetch all the methods $\overline{\texttt{N}}$ defined in the trait T that are not type-checked in C;
• We type-check the set of altered traits $\overline{\texttt{TA}}$ in C producing a set of illegal methods (the except $\overline{\texttt{m}}$ part) which are the methods of $\overline{\texttt{TA}}$ which do not type-check in C;
• We check the condition $\cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}})$ which, intuitively, ensures that every conflict is resolved, and guarantes that the lookup algorithm is a function;
• We build a new set of illegal methods in $\overline{\texttt{T}}$ w.r.t. C, that is, $\mathit{meth}(\overline{\texttt{N}}) \cup (\overline{\texttt{m}} \setminus \mathit{meth}(\overline{\texttt{M}}))$, *i.e.*, the illegal methods of TA $(\mathit{meth}(\overline{\texttt{N}}))$ plus the non-overridden illegal methods from $\overline{\texttt{TA}}$ $(\overline{\texttt{m}} \setminus \mathit{meth}(\overline{\texttt{M}}))$, since overridden ones $\overline{\texttt{M}}$ do not matter anymore.

La règle pour typer une classe est la suivante :

The rule dealing with typing of classes is:

$$\frac{\texttt{K} = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{g}}); \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} \quad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \quad \cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}}) \quad \overline{\texttt{M}} \texttt{ OK IN C} \quad \overline{\texttt{TA}} \texttt{ OK IN C except } \overline{\texttt{m}} \quad \overline{\texttt{m}} \subseteq \mathit{meth}(\overline{\texttt{M}})}{\texttt{class C extends D imports } \overline{\texttt{TA}}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK}}\ (\text{Class·Ok})$$

Table 2.11: La règle de typage pour C dans Jeune Peter / Young Peter's Type Rule for C

Intuitivement cette règle vérifie que toutes les méthodes de la classe sont bien typées et que tous les conflits sont résolus. Cette règle permet également de considérer Jeune Peter comme une extension conservative de Baby Peter, grâce à l'utilisation du symbole $\subseteq$ qui assure la compatibilité dans le cas d'une suite de traits $\overline{\texttt{TA}}$ vide. En quelques mots :
• On énonce le constructeur `K` et ses champs $\overline{\texttt{g}}$;
• On vérifie la condition clé $\cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq meth(\overline{\texttt{M}})$ qui garantit qu'il n'y a pas de conflits et que l'algorithme de liaison dynamique est déterministe ;
• On type toutes les méthodes $\overline{\texttt{M}}$ définies dans `C` ;
• On type l'ensemble des traits altérés $\overline{\texttt{TA}}$ dans `C`, en produisant un ensemble de méthodes illégales $\overline{\texttt{m}}$ (la partie `except` $\overline{\texttt{m}}$), c-à-d. les méthodes de $\overline{\texttt{TA}}$ qui ne se typent pas dans `C` ;
• On vérifie la condition $\overline{\texttt{m}} \subseteq meth(\overline{\texttt{M}})$ qui certifie que les $\overline{\texttt{m}}$ méthodes illégales sont redéfinies avec des méthodes $\overline{\texttt{M}}$ dans la classe `C`.

Intuitively, this rule checks that all the components of the class are well-typed, and that all conflicts are resolved. This type-checking rule ensures that Young Peter is a proper extension of Baby Peter, thanks to both occurrences in the premises of the $\subseteq$ symbol which ensures compatibility whenever $\overline{\texttt{TA}}$ is empty. In a nutshell:
• We fetch the constructor `K` and the fields $\overline{\texttt{g}}$;
• We check the condition $\cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq meth(\overline{\texttt{M}})$ ensuring that every conflict is resolved (see the explanation about (Trait·Ok) above);
• We type-check all methods $\overline{\texttt{M}}$ defined in `C`;
• We type-check the set of altered traits $\overline{\texttt{TA}}$ in `C`, we produce a set of illegal methods $\overline{\texttt{m}}$ (the `except` $\overline{\texttt{m}}$ part), *i.e.*, the methods of $\overline{\texttt{TA}}$ which do not type-check in `C`;
• We check the condition $\overline{\texttt{m}} \subseteq meth(\overline{\texttt{M}})$ ensuring that the $\overline{\texttt{m}}$ illegal methods are overridden with methods $\overline{\texttt{M}}$ of the class `C`.

2.2 Typer les traits séparément / Typeckecking Traits in Isolation

Vu la simplicité de Jeune Peter, les traits peuvent être typés seulement *à l'intérieur d'une classe* et donc pour chaque classe il faut retyper les traits importés (compilation globale). Ce comportement n'est pas compatible avec le principe qui consiste à compiler les traits séparément, comme des interfaces Java. Pour cette raison on étend Jeune Peter avec des Interfaces. On appelle cette extension Ado Peter. Dans Ado Peter, chaque trait sera compilé *une seule fois* (compilation séparée).

Because of the simplicity of Young Peter, traits could be typechecked only *inside a class*, thus they needed to be typechecked *once for every class* (global compilation). This behavior is not compatible with the idea of compiling traits in isolation, like Java interfaces. For this reason, we extend Young Peterwith Interfaces. We call this extension Ado Peter. In Ado Peter, traits need only be typechecked *once and for all* (separate compilation).

2.2.1 Syntaxe / Syntax

La nouvelle syntaxe qui considère les traits comme des types est la suivante :

The new syntax to consider trait-as-types is as follows:

`TL`	::=	`trait T [imports` $\overline{\texttt{TA}}$`]` $\{\overline{\texttt{M}};\overline{\texttt{S}}\}$	Trait Declarations
`I`	::=	`C` | `T`	Types
`S`	::=	`I m(`$\overline{\texttt{I}}$ $\overline{\texttt{x}}$`);`	Signatures

Table 2.12: Syntaxe de Ado Peter / Ado Peter's Syntax

Si on doit considérer les traits comme des types, alors la relation de sous-typage devra comparer non seulement des classes mais aussi des traits. Pour donner une intuition sur la nouvelle règle de sous-typage, on rappelle que, comme pour Java, le typage et le sous-typage de Ado Peter est basé sur les *noms* des types et non sur leurs structures (un type est le nom de la classe ou de l'interface ou du trait). Dans cette extension, les traits sont de vrais types et des règles supplémentaires de sous-typage sont à ajouter. Dans les règles (Sub·Trait) et (Sub·Class·Trait), la fonction *head* donne le nom du trait *en tête* d'une altération. L'idée est que les altérations d'un trait changent le comportement du trait lui-même mais ne changent pas son interface.

In this extension, the subtyping relation does not only compare classes but also traits. To give an intuition about the above rules, we will remind that, as in Java, Ado Peter typing and subtyping is *name-based* (a type is the name of a class or an interface). We intend to stick to this policy in Ado Peter. Since traits are now actual types, the following two rules need to be added, where the function *head* returns the name of the trait which is the *head* of the trait alteration. The rationale is that the alterations do indeed alter the behavioral content of traits, but they do not change their interface.

$$\frac{\texttt{trait T imports } \overline{\texttt{TA}} \ \{\ldots\} \quad \texttt{TA} \in \overline{\texttt{TA}}}{\texttt{T} <: \mathit{head}(\texttt{TA})}\ (\text{Sub·Trait})$$

$$\frac{\texttt{class C extends D imports } \overline{\texttt{TA}} \ \{\ldots\} \quad \texttt{TA} \in \overline{\texttt{TA}}}{\texttt{C} <: \mathit{head}(\texttt{TA})}\ (\text{Sub·Class·Trait})$$

Table 2.13: Les règles de sous-typage de Ado Peter pour les traits comme types / Ado Peter's Extra Subtyping Rules for Traits-as-types

2.2.2 Exemples / Examples

On définit deux traits `TA` et `TB` qui ont des méthodes en commun :

We define two traits that have some methods in common:

```
trait TA {String p(){return this.r()+this.s()+this.q();}}
          String r();
          String s(){return ‘‘Java’’};
          String q();}
trait TB {String r(){return ‘‘Hallo World, my name is’’;
          String s(){return ‘‘Peter’’}}
```

Table 2.14: Deux traits simples vus comme des types en Ado Peter/ Two Simples Ado Peter's trait-as-types

Ces deux traits `TA` et `TB` sont définis une seule fois et peuvent être utilisés et importés plusieurs fois par différentes classes. Le trait `TA` définit la méthode `s` et `p`, qui fait appel, via

Both traits `TA` and `TB` are defined only once and they can be used by any class declaration. Trait `TA` defines method `s` and `p` which on turn requires, via `this`, methods `r`, `s`, and `q`); trait

`this`, aux méthodes `r`, `s` et `q` ; le trait `TB` lui aussi définit les méthodes `r` et `s`. Ces traits peuvent être importés ensemble par une classe de la façon suivante :

`TB` also defines methods `r` and `s`. Those traits can be at once imported in a class declaration as follows:

```
class Presentation extends Object imports TA,TB
 {String ciao(){return this.p()}
  String    s(){return ‘‘Peter with Traits&Interfaces,’’}
  String    q(){return ‘‘I hope you will like me’’}}

(new Presentation()).ciao()
‘‘Hallo World,my name is Peter with Traits&Interfaces,I hope you will like me’’
```

Table 2.15: Une présentation de Ado Peter / Ado Peter's Presentation

et le résultat de l'exécution de ce petit programme introduira Ado Peter de façon sympathique.

and the result of this small program will be a gentle Ado Peter's presentation.

Plusieurs traits peuvent être importés dans une classe ; les conflits entre leurs méthodes communes, définies dans au moins deux des traits importés, doivent être résolus *explicitement* par le programmeur, en renommant ou en excluant les noms des méthodes en conflit dans la classe qui importe les traits ou même à l'intérieur d'un trait.

Multiple traits can be imported by one class, and conflicts between common methods, defined in two or more imported traits, must be resolved *explicitly* by the user, either by aliasing or excluding method names in traits, or by overriding the conflicted methods in the class that imports those traits or in the trait itself.

2.2.3 Un exemple moins trivial avec les traits comme types / A Less Trivial Example with Traits-as-Types

Cet exemple utilise une simple classe de Ado Peter `Integer` qui contient des méthodes algébriques simples comme, `mod` et `times`. Tout d'abord on définit un trait `Convertible` qui est complètement *abstrait* (comme une interface en Java et de la même façon, il n'est utilisé que pour le typage) ; ce trait déclare une méthode `to_int` qui renvoie un entier. Ensuite, on déclare un trait `Hashable` qui importe `Convertible` et invoque la méthode `to_int` comme entrée pour sa fonction de *hashage*. Indépendamment, on définit un trait `Convertible_Pair` qui importe `Convertible` (donc tout `Convertible_Pair` sont aussi `Convertible`) et on implante la méthode `to_int` pour un couple d'objets convertibles.

This example makes use of the simple Ado Peter class `Integer` composed with some simple algebraic methods, *e.g.* `mod` and `times`. First we define a trait `Convertible` which is purely *abstract* (as an interface in Java, we define it only for typing purposes); it requires a method `to_int` producing an integer. Then we declare a trait `Hashable` that imports `Convertible`, and uses the `to_int` method as an input for its hashing function. Independently, we define a trait `Convertible_Pair` that imports `Convertible` (thus every `Convertible_Pair` is also `Convertible`) and we implement the method `to_int` for a pair of two convertible objects.

```
trait Convertible {Integer to_int();}
trait Hashable imports Convertible {Integer hash() {... this.to_int() ...}}
trait Convertible_Pair imports Convertible {
  Integer to_int()
  {return this.fst().to_int().plus(this.snd().to_int().times(this.offset()));}
  Convertible fst();
  Convertible snd();
  Integer offset();}
trait My_String imports Hashable { }
```

Table 2.16: Les traits `Convertible`, `Hashable`, `Convertible_Pair` et `My_String` en Ado Peter / Ado Peter's Traits `Convertible`, `Hashable`, `Convertible_Pair`, and `My_String`

On pourra donc définir une extension `H_Integer` de la classe `Integer` avec une méthode `hash` qui récupère la valeur hash de l'entier considéré.

It allows then to define an extension `H_Integer` of the class `Integer` with a method `hash` to get a hash value of the considered integer.

```
class H_Integer extends Integer imports Hashable { ;
  H_Integer()      {super();}
  Integer to_int() {return this;} }
```

Table 2.17: La classe `H_Integer` en Ado Peter / Ado Peter's Class `H_Integer`

Finalement, on peut définir nos chaînes comme des listes de caractères. Les chaînes sont des sous-classes du trait `My_String`, qui est une interface équivalente à `Hashable` (dans une implantation réaliste il devrait plutôt être un sous-type strict de `Hashable`). Donc, on a deux classes qui construisent des chaînes, `Null_String`, une classe avec un seul élément et `Cons_String` qui est intrinsèquement un couple d'un caractère et d'une chaîne. La classe `Cons_String` importe `Convertible_Pair` pour implanter la méthode `to_int` qui est demandée par `My_String` (comme trait importé par `Hashable`).

Finally, we define our strings as lists of characters. The strings are indeed subclasses of a trait `My_String`, which is an interface equivalent to `Hashable` (in real life it would be a strict subtype of `Hashable` though). As such, we have two classes that build strings, namely `Null_String`, *i.e.* a single element class, and `Cons_String` which is intrinsically a pair of a character and a string. The class `Cons_String` imports `Convertible_Pair` to implement the method `to_int` which is required by `My_String` (as a trait importing `Hashable`).

```
class My_Char extends Objects imports Convertible { ;
  int me;
  My_Char(Integer c) {super();this.me=c.mod(new Integer(256));}
  Integer   to_int() {return this.me;} }
class Cons_String extends Objects imports My_String Convertible_Pair {
  My_Char     head
  My_String   tail;
  Cons_String(My_Char c,My_String s)
                    {super();this.head=c; this.tail=s;}
  Convertible fst() {return this.head;}
  Convertible snd() {return this.tail;}
  Integer  offset() {return new Integer(256);} }
class Null_String extends Objects imports My_String {
  Null_String() {super();}
  Integer to_int() {return new Integer(0);} }
```

Table 2.18: Les classes `My_Char`, `Cons_String` et `Null_String` en Ado Peter / Ado Peter's Classes `My_Char`, `Cons_String`, and `Null_String`

Finalement, on peut définir une classe `Array` qui peut être utilisée comme un tableau ou une table associative. Avec la fonction de hashage sur les chaînes (ou celle de hashage sur les entiers) la classe peut être utilisée comme une table de hashage, comme montré dans l'exemple. En plus, `H_Integers`, `Cons_String` et `Null_String` auront non seulement en commun le type `Hashable` mais aussi une implantation commune de la méthode `hash` héritée via traits.

Finally, we can define a class `Array`, which can be used as an array or an association table. Together with the hashing function of strings (or that of hashable integers) it may be used as a hash table, as showed in the example. Moreover, `H_Integers`, `Cons_String`, and `Null_String` do not only share the `Hashable` type, but they also have a common implementation of the method `hash` yielded by trait inheritance.

```
class Array extends Objects imports H_Integers {
  ...
  Object index(Integer i) {...}
  Object assoc(Integer i) {...}}

(Array)
 (new Array(...).index(new Cons_String(...).hash())).assoc(new Cons_String(...))
```

Table 2.19: La classe `Array` en Ado Peter / Ado Peter's class `Array`

2.2.4 Règles clés pour le typage / Key Type-checking Rules

La règle (Trait·Ok) vérifie que tous les composants d'une classe et d'un trait sont bien typés et que tous les conflits sont résolus ; la règle construit une liste des méthodes qui sont requises mais non implantées dans le trait. Le

The (Trait·Ok) rule check that all the components of the class and of the trait are well-typed, and that all conflicts are resolved; the trait rule also builds the list of methods that are required but not provided in the trait. We in-

jugement $\overline{\texttt{S}}$ `OK IN T` signifie que les méthodes avec signature $\overline{\texttt{S}}$ ne soulèvent pas un conflit de type dans `T`; il sert à vérifier que les traits importés sont compatibles (par exemple, une classe ne peut pas importer un trait avec une méthode `m` qui rend un entier et un autre trait avec la même méthode `m` qui rend une chaîne de caractères).

troduce the judgment $\overline{\texttt{S}}$ `OK IN T` basically meaning that the methods whose signature are in $\overline{\texttt{S}}$ do not raise a typing conflict with `T`; this judgment is used to check that the imported traits are compatible (a class cannot import a trait with a method `m` returning an integer and another trait with a method `m` returning a string, for instance).

$$\frac{\cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}}) \qquad \overline{\texttt{M}}\ \texttt{OK IN T} \qquad \overline{\texttt{TA}}\ \texttt{OK requires}\ \overline{\texttt{S}}' \qquad (\mathit{sig}(\overline{\texttt{TA}}) \cup \overline{\texttt{S}})\ \texttt{OK IN T}}{\texttt{trait T imports}\ \overline{\texttt{TA}}\ \{\overline{\texttt{M}};\overline{\texttt{S}}\}\ \texttt{OK requires}\ (\overline{\texttt{S}} \cup \overline{\texttt{S}}') \setminus \mathit{meth}(\texttt{T})}\ (\text{Trait·Ok})$$

Table 2.20: La nouvelle règle (Trait·Ok) dans Ado Peter avec les traits vus comme des types / Ado Peter's New Rule (Trait·Ok) with Traits-as-types

Dans une «coquille de noix» :
• On type l'ensemble de traits altérés $\overline{\texttt{TA}}$ en produisant un ensemble de méthodes qui restent à implanter (la partie `requires` $\overline{\texttt{S}}'$) ;
• On vérifie la condition $\cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}})$ qui garantit que chaque conflit est résolu et que l'algorithme de liaison dynamique est une fonction ;
• On type les signatures des méthodes $\overline{\texttt{M}}$ dans `T` ;
• On type les signatures des méthodes dans $\overline{\texttt{TA}}$ et $\overline{\texttt{S}}$; tous les traits $\overline{\texttt{TA}}$ et la signature $\overline{\texttt{S}}$ doivent être compatibles dans le trait `T`, c-à-d. qu'ils peuvent être composés deux à deux sans aucun conflit de typage (par exemple, des conflits de typage peuvent avoir lieu entre une méthode implantée et une méthode abstraite) ;
• L'ensemble des méthodes qui restent à implanter dans `T` est donc l'ensemble $\overline{\texttt{S}}'$ de toutes les signatures des méthodes qui restent à implanter dans les traits altérés importés $\overline{\texttt{TA}}$ plus l'ensemble $\overline{\texttt{S}}$ déclaré dans `T` à l'exception de toutes les méthodes qui sont définies (c-à-d. implantées) dans `T`. On observe que le programmeur en Ado Peter peut définir plus de signatures de méthodes dans $\overline{\texttt{S}}$ que ce qui lui est vraiment nécessaire; en d'autres mots : si une méthode `m` a été déclarée dans $\overline{\texttt{S}}$ mais que son comportement est déjà présent dans un trait importé, alors le système de types se comporte exactement comme si cette méthode `m` n'apparaissait pas dans $\overline{\texttt{S}}$ (les types de deux déclarations doivent être égaux).

In a nutshell:
• We typecheck the set of altered traits $\overline{\texttt{TA}}$ producing a set of required methods (the `requires` $\overline{\texttt{S}}'$ part);
• We check the condition $\cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}})$ ensuring that every conflict is resolved, and guaranteeing that the lookup algorithm is a function;
• We typecheck the methods $\overline{\texttt{M}}$ inside `T`;
• We typecheck the method signatures in $\overline{\texttt{TA}}$ and $\overline{\texttt{S}}$; all the traits $\overline{\texttt{TA}}$ and the signature $\overline{\texttt{S}}$ must be compatible in the trait `T`, *i.e.* they can be pairwise composed without any conflict in the method types (for example, type conflicts may arise between an existing and an abstract method);
• Consequently, the set of required methods in `T` is the set $\overline{\texttt{S}}'$ of all methods signature required by the imported trait alterations $\overline{\texttt{TA}}$ plus the set $\overline{\texttt{S}}$ declared in `T` except all the methods that are defined (*i.e.* implemented) in `T`. It is worth noticing here that an Ado Peter's programmer can define more methods' signatures in $\overline{\texttt{S}}$ than what is really needed; in other words: if a method `m` has been declared in $\overline{\texttt{S}}$ but its behavior is already present inside an imported trait, then the type system behaves exactly as if `m` did not appear in $\overline{\texttt{S}}$ (of course the types of both declarations must be equal).

Il reste à présenter la règle (Class·Ok) :

We can now introduce the type rule (Class·Ok):

$$\frac{\texttt{K} = \texttt{C}(\overline{\texttt{J}}\ \overline{\texttt{g}}, \overline{\texttt{I}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{g}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\} \quad \mathit{fields}(\texttt{D}) = \overline{\texttt{J}}\ \overline{\texttt{g}} \quad \cap\overline{\texttt{TA}} \setminus \diamond\overline{\texttt{TA}} \subseteq \mathit{meth}(\overline{\texttt{M}}) \quad \mathit{sig}(\overline{\texttt{TA}})\ \texttt{OK IN C} \quad \overline{\texttt{M}}\ \texttt{OK IN C} \quad \overline{\texttt{TA}}\ \texttt{OK requires}\ \overline{\texttt{S}} \quad \mathit{meth}(\overline{\texttt{S}}) \subseteq \mathit{meth}(\texttt{C})}{\texttt{class C extends D imports}\ \overline{\texttt{TA}}\ \{\overline{\texttt{I}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK}}\ (\text{Class·Ok})$$

Table 2.21: La nouvelle règle (Class·Ok) dans Ado Peter avec les traits vus comme types / Ado Peter's New Rule (Class·Ok) with Traits-as-types

Dans une «coquille de noix» :
• On énonce le constructeur K et le champ $\overline{\texttt{g}}$;
• On vérifie la condition pour la résolution de conflits $\cap\overline{\texttt{T}} \setminus \diamond\overline{\texttt{T}} \subseteq \mathit{meth}(\overline{\texttt{M}})$;
• On type toutes les méthodes $\overline{\texttt{M}}$ définies dans C ;
• On type l'ensemble des traits $\overline{\texttt{TA}}$, en produisant un ensemble d'interfaces requises $\overline{\texttt{S}}$ (la partie `requires` $\overline{\texttt{S}}$) par $\overline{\texttt{TA}}$, c-à-d. les méthodes de $\overline{\texttt{TA}}$ qui ne sont pas typables dans $\overline{\texttt{TA}}$ lui-même ;
• On type la signature de $\overline{\texttt{TA}}$. Comme pour la règle (Trait·Ok), cela vérifie que tous les traits $\overline{\texttt{TA}}$ sont compatibles dans la classe C ;
• On vérifie la condition $\mathit{meth}(\overline{\texttt{S}}) \subseteq \mathit{meth}(\texttt{C})$. Cette condition signifie que chaque classe doit fournir une implantation de toutes les méthodes, afin de pouvoir créer de vrais objets «vivants» avec l'expression `new C(...)`. Plus en détail, $\mathit{meth}(\overline{\texttt{S}})$ est la liste de tous les noms des méthodes requises par les traits importés et qui seront implantées dans la classe elle-même, dans sa classe parente ou dans un trait importé. Donc, on vérifie que chaque fois qu'une méthode m est attendue pour C, elle est aussi implantée ou héritée dans C.

In a nutshell:
• We fetch the constructor K and the fields $\overline{\texttt{g}}$;
• We check the condition for conflict resolution $\cap\overline{\texttt{T}} \setminus \diamond\overline{\texttt{T}} \subseteq \mathit{meth}(\overline{\texttt{M}})$;
• We typecheck all methods $\overline{\texttt{M}}$ defined in C;
• We typecheck the set of traits $\overline{\texttt{TA}}$, producing a set of interfaces $\overline{\texttt{S}}$ required by $\overline{\texttt{TA}}$ (the `requires` $\overline{\texttt{S}}$ part), *i.e.*, the methods of $\overline{\texttt{TA}}$ which do not typecheck in $\overline{\texttt{TA}}$ itself;
• We typecheck the method signatures in $\overline{\texttt{TA}}$: as for the rule (Trait·Ok), this verify that all the traits $\overline{\texttt{TA}}$ are compatible in the class C;
• We check the condition $\mathit{meth}(\overline{\texttt{S}}) \subseteq \mathit{meth}(\texttt{C})$. It means that the classes have to provide all the necessary methods in order to be able to instantiate "live" objects via the expression `new C(...)`. Specifically, $\mathit{meth}(\overline{\texttt{S}})$ is the list of all method names that are required by the imported traits and that are being implemented either inside the class, in the superclass, or inside some other trait. As such, we ensure that every time a method m is expected for C then it is also implemented in C or inherited by C.

Sources d'inspiration / Inspiration Sources

Ce chapitre est essentiellement une présentation pédagogique de deux articles que j'ai écrits avec Arnaud sur l'ajout des traits et des traits vus comme des interfaces dans Featherweight Java [1,2]. Le lecteur intéressé par les traits typés trouvera son bonheur dans les articles de Sophia sur Chai [SD05], de John et Kathleen sur Moby [FR04], le langage Scala de Martin [Sca07] et le langage Fortress de Sun [ACL+07].

This chapter is essentially a pedagogical presentation of the two papers I have written with Arnaud that add typed traits and traits-as-interfaces to Featherweight Java[1,2]. The interested reader in typed trait can take a look at the Sophia's Chai language [SD05], and John and Kathleen's Moby languages [FR04], Martin's Scala [Sca07] and Sun's Fortress [ACL+07].

Part II

L'homme Peter (métier) / Peter, the Man (Expertise)

Chapter 3

Peter devient un spécialiste / Peter Becomes a Specialist

Ce chapitre est dédié à Kim, pour sa relation de «matching» qui a marqué un bon nombre de mes «nuits blanches» et de mes articles sur la programmation objet.

This chapter is dedicated to Kim for his matching relation that has influenced a large part of my "wake nights" and my papers on OOP.

3.1 Spécialisation des types des méthodes / Mytype Method Specialization

L'héritage est une technique puissante pour la réutilisation de code ; les méthodes qui sont définies dans les classes parentes n'ont pas besoin d'être redéfinies dans les sous-classes et elles peuvent être appelées. Cela fonctionne très bien dans les langages non typés ou typés dynamiquement comme Smalltalk, mais beaucoup moins bien dans les langages avec du typage statique, comme Java ou Peter. La technique hérite non seulement le corps des méthodes mais aussi leurs signatures et cela n'est pas forcement ce qu'un programmeur souhaite car, dans l'analogie des classes comme des types, on voudrait spécialiser quelque type des méthodes héritées. Le problème peut être simplement expliqué en utilisant deux «classes classiques» :

Inheritance is a powerful technique for code reusability; methods defined in superclasses need not be redefined in subclasses and they are fully re-usable. This works especially fine in untyped or dynamically-typed languages *à la* Smalltalk, but lesser when dealing with statically-typed languages like Java or Peter. The inheritance mechanism allows to have inheritance not only of method bodies but also of their signatures. Unfortunately, this is not always what a programmer really wants, since, in the classes-as-types analogy, some inherited methods are amenable of type-specialization. The problem can be easily made clear using two "classic classes":

```
class Point extends Object
{int x;int y;
 Point(int x,int y){super();this.x = x;this.y = y;}
 Point move(int x,int y){return new Point(thix.x + x,this.y + y);}}

class Pix_Point extends Point
{bool pix;
 Pix_Point(int x,int y,bool pix){super(x,y);this.pix = pix;}

    Point p = new Point(0,0);                          Let p is of class Point...
Pix_Point q = new Pix_Point(1,1,true);                 ...and q is of class Pix_Point
... p.move(2,2).x                  Typable... p.move(2,2) has type Point so x is known
... q.move(3,3).pix     Untypable... q.move(3,3) has still type Point, so pix is unknown
```

Table 3.1: Les classes `Point` et `Pix_Point` en Baby Peter / Baby Peter's Classes `Point` and `Pix_Point`

La faiblesse du système de types actuel de Baby Peter est que les types des méthodes héritées ne se spécialisent pas lorsqu'on les hérite. Dans l'exemple précédent, si on invoque une méthode `move` sur un objet de type statique `Pix_Point`, alors son type statique est `Point`, ce qui représente un non-sens car l'objet «vivant» est un véritable `Pix_Point`.
En d'autres termes : la méthode `move` est héritée par la sous-classe : elle ne produira pas des erreurs à l'exécution, mais le type de son résultat reste lié à `Point` ce qui est un non-sens car l'objet «vivant» est un vrai `Pix_Point`. Dans cet exemple on n'a que `Pix_Point<:Point` et que le slogan «*l'héritage engendre du sous-typage*» représente l'essence de la programmation à classes. Malheureusement, on verra dans un instant que cela est pas vrai lorsqu'on introduit une nouvelle catégorie des méthodes...

The weakness of the current Baby Peter's type system is that types of the inherited methods are not specialized when inherited. In the previous example, if we send a `move` method to an object of static type `Pix_Point`, then the static result type turns out to be `Point` which does not make any sense because the "live" object is a genuine `Pix_Point`.
In other words: the method `move` is inherited in the subclass and runs correctly, while its result type still remains *anchored* to the type of the superclass. This is, of course, a weakness of the type system. In this simple example we have that `Pix_Point<:Point` and that the simple slogan *"Inheritance is subtyping"* is true and it represents the essence of class-based typed programming. Unfortunately, as we will see later this is not true at all when a special kind of method is introduced...

3.1.1 Pourquoi les sous-classes ne produisent pas de sous-types ? / Why Inheritance is not Subtyping?

Un simple ajout aux classes (qu'on vient de montrer) des méthodes typiquement appelés *binaires* détruit la sûreté du système de types, si on considère l'équation sous-classes = sous-types. Intuitivement, une méthode binaire est une méthode qui prend comme paramè tre un objet de la même classe de l'objet qui reçoit le message : des exemples classiques de méthodes binaires sont les méthodes qui vérifient l'égalité structurelle entre l'objet récepteur de la méthode et l'objet argument

The introduction of the so-called *binary* methods to the previous two classes, destroys the type safety of the system: this breaks the property that subclasses always generate subtypes. Intuitively, a binary method takes as input parameter an object of the same class of the receiver: typical examples are the *equality* methods that check whether the receiver is structurally equal to another object of the same class. Without a subtyping relation (`C<:D` means that every object of type `C` can be used

déclaré de la même classe que le récepteur. Sans la règle de sous-typage (qui dit que si `C <: D`, alors tout objet de type `C` peut être utilisé dans n'importe quel contexte qui attend un objet de type `D`) la présence de méthodes binaires ne détruit pas la sûreté du système de types. Le *caveat* est que le sous-typage *est* une composante essentielle de la programmation objets et il est impossible de concevoir un tel paradigme sans le sous-typage.
Mais avançons lentement pour présenter les choses dans l'ordre et clairement. La table suivante présente les classes `Point` et `Pix_Point` améliorées avec une méthode binaire qui vérifie l'égalité entre objets. Malheureusement, la formulation actuelle du système de types de Baby Peter ne typera pas `Pix_Point` car elle redéfinit la méthode `equal` en lui modifiant sa signature.

in any context where an object of type `D` can be used), the presence of binary methods in classes do not affect type safety.

The *caveat* is that subtyping *is* a fundamental part of the object-oriented paradigm and it is almost impossible to get rid of it.

Let's begin slowly in order to make the problem clear. The next table shows an improvement of classes `Point` and `Pix_Point` with a binary method verifying the equality between points. Unfortunately, the actual formulation of the Baby Peter's type system makes class `Pix_Point` untypable because of the changed signature of the `equal` method.

```
class Point extends Object
{int x;int y;
 Point(int x,int y){super();this.x = x;this.y = y;}
 Point move(int x,int y){return new Point(thix.x + x,this.y + y);}
     bool equal(Point p){return this.x == p.x && this.y == p.y;}}

class Pix_Point extends Point
{bool pix;
 Pix_Point(int x,int y,bool pix){super(x,y);this.pix = pix;}
 bool equal(Pix_Point p){return this.x==p.x && this.y==p.y && this.pix==p.pix;}}
```

Table 3.2: Les classes (non typables en Baby Peter) `Point` et `Pix_Point` avec la méthode `equal` / Baby Peter's Untypable Classes `Point` and `Pix_Point` with the `equal` Method

Si on applique la définition de la fonction *override* de Baby Peter, on pourra constater que la propriété d'*invariance* lorsqu'on redéfinit une méthode n'est pas respectée, c-à-d. le prédicat *override* de la règle (Meth·Ok·Class) (voir Table 1.3) est faux :

By looking at the definition of the Baby Peter *override* function, one can notice that the *invariance* property of the overridden method is not respected, *i.e.* the *override* predicate in the (Meth·Ok·Class) rule (voir Table 1.3) is false:

$$override(\texttt{equal}, \texttt{Point}, \texttt{Pix\_Point -> bool}) = \texttt{false}$$

car `Pix_Point -> bool` est différent de `Point -> bool`[1]. Si le système de types de Baby Peter acceptait ces deux classes avec le sous-typage `Pix_Point<:Point`, alors il ne serait pas difficile de faire «exploser» à l'exécution un programme bien typé en utilisant une simple classe auxiliaire `Spoof` (cela a été montré il y a plusieurs années par Kim [Bru02]) :

since `Pix_Point -> bool` is different from `Point -> bool`[2]. If the Baby Peter's type system accepted these two classes and assumed `Pix_Point<:Point`, then it would be not difficult to get a crash at run-time for a well-typed program. It would be enough to use a simple spoofing class `Spoof` (this was showed many years ago by Kim [Bru02]):

1 Dans autres systèmes de types orienté-objets on demande que `Pix_Point -> bool<:Point -> bool`, ce qui implique, par contravariance du constructeur de type flèche, `Point<:Pix_Point` qui est trivialement faux.

2 In other object-oriented type systems we require `Pix_Point -> bool<:Point -> bool`. By the contravariance of the arrow constructor, that implies `Point<:Pix_Point`, which is obviously false.

```
class Spoof extends Objects
{;Spoof(){super();}
 bool crash(Point p,Point q){return p.equal(q);}}

Pix_Point p = new Pix_Point(1,1,true);                          p is a Pix_Point
    Point q = new Point(1,1);                                      q is a Point
    Spoof s = new Spoof();                                s is a spoofing instance of Spoof
s.crash(p,q);   method equal of Pix_Point is called with argument q and q.pix will crash
```

Table 3.3: L'explosion d'un programme non typable en Baby Peter avec une méthode binaire / The Crash of an Ill-typed Baby Peter's program with a Binary Method

3.1.2 La solution : les types qui se spécialisent / The Solution: types that Specialize

Pour faire en sorte que les types des méthodes (et en particulier des méthodes binaires) puissent se spécialiser avec l'héritage, on introduit le nouveau type `Mytype`. Bien sûr l'ajout de cette méta-variable de type comportera le changement complet du système de types de Baby Peter car la discipline induite par `Mytype` est très proche de celle des types récursifs et ou du «*bounded polymorphism*». On appelle cette extension Sharp Peter.

Récrivons les deux toutes premières classes précédentes avec ce nouveau type en Sharp Peter :

In order to specialize with inheritance the type of methods (and in particular binary methods), we introduce the new type `Mytype`. The introduction of a simple type metavariable has a tremendous impact on the whole Baby Peter's type system, since the type discipline has strong similarities with recursive-types and bounded polymorphism. We call this extension Sharp Peter.

Let's try to recast in Sharp Peter the very first two classes with this new type:

```
class Point extends Object
{int x;int y;
 Point(int x,int y){super();this.x=x;this.y=y;}
 Mytype move(int x,int y){return new Point(thix.x+x,this.y+y);}}

class Pix_Point extends Point
{bool pix;
 Pix_Point(int x,int y,bool pix){super(x,y);this.pix=pix;}

   Point p = new Point(0,0);                                  Let p is of class Point...
Pix_Point q = new Pix_Point(1,1,true);                  ...and q is of class Pix_Point
... p.move(2,2).x                  Typable... p.move(2,2) has type Point so x is known
... q.move(3,3).pix Typable... q.move(3,3) has now type Pix_Point, so pix is well-known
... p=q                                    Typable since Pix_Point is a subtype of Point
```

Table 3.4: Les classes `Point` et `Pix_Point` en Sharp Peter / Sharp Peter's Classes `Point` and `Pix_Point`

L'utilisation du nouveau type `Mytype` en position résultat (ou *covariante*) dans la méthode `move` est compatible dans le nouveau système de types spécialisé de Sharp Peter avec la relation de sous-typage : on peut facilement observer une augmentation importante d'expressivité du système de types.

The use of the new type `Mytype` as result (*covariantly*) in method `move` is compatible in the new specialized Sharp Peter's type system with subtyping: we can easily see a remarkable increase of expressivity of the new type system.

Malheureusement, une occurrence *contravariante* de `Mytype` dans une signature d'une méthode, implique encore qu'une sous-classe n'est pas un sous-type ; si `Pix_Point<:Point` alors le programme suivant serait bien typé mais il produirait une erreur à l'exécution :

Unfortunately, a *contravariant* occurrence of `Mytype` in a method signature, still implies that a subclasses does not generate subtypes; if `Pix_Point<:Point`, then the following well-typed program will crash at run-time:

```
class Point extends Object
{int x;int y;
 Point(int x,int y){super();this.x=x;this.y=y;}
 Mytype move(int x,int y){return new Point(thix.x+x,this.y+y);}
     bool equal(Mytype p){return this.x==p.x && this.y==p.y;}}

class Pix_Point extends Point
{bool pix;
 Pix_Point(int x,int y,bool pix){super(x,y);this.pix=pix;}
 bool equal(Mytype p){return this.x==p.x && this.y==p.y && this.pix==p.pix;}}

class Spoof extends Objects
{;Spoof(){super();}
 bool crash(Point p,Point q){return p.equal(q);}}

Pix_Point p = new Pix_Point(1,1,true);
    Point q = new Point(1,1);
   Spoof  s = new Spoof();
... s.crash(p,q);            Method equal of Pix_Point is called and q.pix will crash.
```

Table 3.5: Comment faire «exploser» un programme bien typé en Sharp Peter avec une méthode binaire et du sous-typage / How to "Crash" a Well-typed Sharp Peter's Program with a Binary Method and Subtyping

La morale de `Mytype` est qu'elle peut être utilisée, d'une façon sûre, dans un système de types où la relation de sous-typage est *désactivée*.

The moral of introducing `Mytype` is that it can be safely used only in type systems without subtyping.

3.1.3 Règles clés pour le typage / Key Type-checking Rules

Le système de types $\vdash_{\mathsf{sharp}}$ de Sharp Peter est une extension du système de Baby Peter $\vdash_{\mathsf{baby}}$. Comme dit précédemment, il est basé sur la relation de *matching* de Kim [Bru02]. Les règles principales du système sont :

The type system $\vdash_{\mathsf{sharp}}$ for Sharp Peter is an extension of the Baby Peter's type system $\vdash_{\mathsf{baby}}$. As said before, it is based on the Kim's Matching relation [Bru02]. The main type rules are:

$$\frac{\texttt{CT(C)} = \texttt{class C extends D } \{\ldots\}}{\Gamma \vdash_{\textsf{sharp}} \texttt{C <\# D}}\ (\text{Match·Class}) \qquad \frac{\texttt{Mytype <\# C} \in \Gamma}{\Gamma \vdash_{\textsf{sharp}} \texttt{Mytype <\# C}}\ (\text{Match·Var})$$

$$\frac{\Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \quad \texttt{m} \in \mathit{meth}(\texttt{C}_0) \quad \mathit{mtype}(\texttt{m}, \texttt{C}_0) = \overline{\texttt{D}} \to \texttt{C} \quad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{D}}}{\Gamma \vdash_{\textsf{sharp}} \texttt{e}_0\texttt{.m}(\overline{\texttt{e}}) : \texttt{C}[\texttt{C}_0/\texttt{Mytype}]}\ (\text{Type·Call})$$

$$\frac{\overline{\texttt{x}}{:}\overline{\texttt{C}}, \texttt{Mytype <\# D}, \texttt{this:Mytype} \vdash_{\textsf{sharp}} \texttt{e} : \texttt{E}[\texttt{D}/\texttt{Mytype}] \quad \texttt{m} \in \mathit{meth}(\texttt{D}) \ \mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{E})}{\texttt{E m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{return e;}\}\ \texttt{OK IN C}}\ (\text{Meth·Ok·Class})$$

Table 3.6: Les règles principales de typage de Sharp Peter/ Sharp Peter's Main Typing Rules

Dans une «coquille de noix» :
• (Match·Class) Cette règle, qui est la clé de la relation de matching de Kim, dit que C possède au moins les mêmes méthodes de D avec les mêmes signatures (nous observons que chaque occurrence de Mytype aura une sémantique qui variera en fonction de son «ancrage» à une classe) ;
• (Match·Var) Cette règle dit que la métavariable Mytype correspond à une classe C, si cette «contrainte de correspondance» a été déclarée dans le contexte de typage ;
• (Type·Call) Cette règle est pratiquement la même que celle de Baby Peter, avec l'exception que le sous-typage est interdit ; les types des paramètres actuels seront exactement les mêmes que ceux des paramètres formels ; les types du récepteur seront compatible avec une classe qui contient au moins la méthode qu'on appelle ; le type du résultat est celui déclaré dans la signature, où chaque occurrence de Mytype sera remplacée par le type statique du récepteur ;
• (Meth·Ok·Class) Cette règle explique comment les contraintes de «matching» sont utilisées ; le type de this est Mytype et cette méta variable doit être compatible avec le type statique D du récepteur du message, contenu dans le contexte de typage et qui contient au moins la méthode m que l'on est en train d'appeler ; le type du résultat sera celui déclaré dans la signature, où chaque occurrence de Mytype sera remplacé par le type statique du récepteur.

In a nutshell:
• (Match·Class) This rule, which is the core of the Kim's matching relation, says that C has at least the same methods of D with the same signatures (note that every occurrence of Mytype will have a semantics depending on which class is "anchored" to);
• (Match·Var) This rule says that the metavariable Mytype matches with a class C in case this *matching-constraint* is declared in the context Γ;
• (Type·Call) This rule is almost the one of Baby Peter, with the exception that subtyping is no more allowed; the types of the actual parameters must *match* exactly with the formal parameters; the type of the receiver must match with a class containing the method we are calling; the result type of the call is the one declared in the signature, where every free occurrence of Mytype will be replaced with the static type of the receiver itself;
• (Meth·Ok·Class) This rule shows the constrained nature of the type system; the type of this will be the type-variable Mytype and this type-variable is "match-constrained" with the static type D of the receiver containing at least the method m; the final argument type of the method will be the one declared in the signature, where every free occurrence of Mytype will be replaced with the concrete type of the receiver itself.

Sources d'inspiration / Inspiration sources

Ce chapitre a été inspiré par les articles de Kim sur la spécialisation de type à travers le «matching» (quasiment tout est repris dans son

This chapter has been influenced by Kim's papers on type specialization via matching (almost all the story in his excellent book [Bru02])

excellent livre [Bru02]) et dans un certain nombre d'articles que j'ai publié sur le «matching» appliqués aux langages objets purs, formalisés dans le Lambda Calcul des Objets de Kathleen, Furio et John [FHM94] et dans le Calcul des Objets Primitifs de Martin et Luca [AC96]. Mes papiers les plus pertinents sont [8,9,10,32,33,34,35,38,39,40,41,42,52,58] : le plus pédagogique est le [36].

and by a certain numbers of papers I have published on matching applied on object-based languages, semantically modeled in the Kathleen, Furio, and John's Lambda Calculus of Objects [FHM94] and in the Martin and Luca's Object Calculus [AC96]. My most pertinents papers are [8,9,10,32,33,34,35,36,38,39,40,41,42,52,58]: the most pedagogical is [36].

Chapter 4

Peter s'envole / Peter can Fly

Ce chapitre est dédié au «chercheur de talents» Pierre, pour son invitation à postuler à l'École Normale Supérieure de Lyon, ma première île française que je n'oublierai jamais.

This chapter is dedicated to the "talent scout" Pierre, for his kind invitation to apply to the École Normale Supérieure de Lyon, my first french island that I will never forget.

4.1 Un objet qui échappe à sa classe / An Object Escapes from its Class

Un vieux chien pourrait sembler apprendre de nouveaux tours si dans sa jeunesse on lui avait enseigné un tour «d'auto-apprentissage».

An old dog could appear to learn new tricks if in his youth he had been taught a self-extension trick.

La richesse, la rentabilité ou simplement le succès d'un langage à classes est souvent mesuré en termes de simplicité de sa syntaxe et de richesse de ses concepts orientés objets, comme «l'encapsulation», le polymorphisme, l'héritage, etc. Un dogme des langages à classes est que les objets sont créés à partir d'une classe et que l'héritage se place au niveau des classes et non des objets. Chaque fois qu'un objet est créé, il appartient à sa classe pendant toute sa «vie» ; en d'autres termes : «*il ne peut pas s'échapper de sa classe !*» Ce choix a quelques avantages mais aussi quelques défauts. Le plus gros avantage est que la taxonomie statique de la hiérarchie des classes est très précise et, à tout moment, on peut connaître la classe statique des objets qui sont dans la pile (l'arène virtuelle de l'exécution). Malheureusement, la hiérarchie des classes peut devenir très lourde, car il faut ajouter une sous-classe à chaque fois que l'on veut ajouter une variable ou une

The richness or the usability or simply the success of a class-based object oriented language is often measured by the simplicity of its syntax and the richness of the pure object-oriented features (encapsulation, polymorphism, inheritance, etc.). A dogma of the statically-typed class-based languages it that objects are created by class instantiation, and inheritance takes place at the class level. Once the object is created, it "belongs" to its class during all its life; in other words: *"it cannot escape from its class!"* This choice has pros and cons. The big advantage is that the static taxonomy of the class hierarchy is very precise and at any moment we can know the class of objects playing in the run-time arena. Unfortunately, sometimes the class-hierarchy become clumsy, since we must add a subclass any time we want to introduce a slot or a method in an object that shares almost the same behavior and shape of the ones belonging to its direct superclass.

méthode dans un objet qui partage pratiquement les mêmes comportements et le même état des objets qui appartiennent à sa super-classe directe.

Donc, il est impossible d'ajouter une nouvelle classe et de créer à l'exécution de nouveaux objets de cette classe ; la règle est d'arrêter l'exécution, d'ajouter une sous-classe, de recompiler et de re-exécuter. Par contre, il existe plusieurs cas où on voudrait re-classifier un objet car, à un moment de sa vie, il a acquis de nouvelles variables, il a appris de nouvelles méthodes ou il voudrait reimplanter une méthode qu'il possède déjà. Cette thématique de re-classification est bien connue dans le domaine des bases de données persistantes où, par exemple, un étudiant en thèse devient docteur, puis chercheur, puis père et, juste le dimanche, joueur de golf (voir l'excellent papier de Elisa e Giovanna [BG95]). Ce dynamisme ne s'adapte pas aux langages typés statiquement et à classes *à la* Java et Peter.

As such, adding new class and objects at runtime is forbidden; the rule of thumb is to stop the program, add sub-classes, recompile and re-run again. However, there are many cases where we would like to reclassify an object just because at some point of its life it has acquired new slots or it has learned new methods or just because he knows how to reimplement one method it already has. Such reclassification needs are quite well known in permanent databases where, *e.g.* a student becomes a Ph.D., then a researcher, then a father, and, just on Sunday, a golf player (See the excellent paper by Elisa and Giovanna [BG95]). This dynamicity do not match with statically-typed class-based languages *à la* Java and Peter.

Dans beaucoup de situations, en programmation comme dans la vie de tous les jours, on aurait intérêt à manipuler des objets qui changent de comportement lorsqu'ils exécutent un message (ils répondent à une stimulation extérieure). Considérons, par exemple, les cas suivants :

There are plenty of situations, both in programming, and in real life, where it would be convenient to have objects which modify their interface upon an execution of a message (they react to an outside stimulation). Consider for instance the following situations:

• comme déjà dit, dans les langages à classes typés statiquement, on ne peut modifier la structure d'une classe que de façon statique. Si on doit ajouter une nouvelle méthode ou une nouvelle variable à une instance d'une classe, on est obligé de modifier la classe et de la recompiler ; la modification sera visible dans *toutes les instances* et, donc, la sémantique du programme va changer (avec un gâchis de mémoire dans le cas d'ajouts de variables). Alternativement, on pourrait ajouter une sous-classe *ad hoc* pour prendre en compte la modification *in situ*. Si une classe pouvait avoir une méthode qui lui permettait de s'auto-étendre, alors seules les instances de la classe qui exécuteraient cette méthode changeraient leur sémantique (ou alloueraient de la mémoire), sans besoin de recompiler le programme tout entier ;

• in typed class-based languages we can modify the structure of the class only statically. If we need to add a new method or add a new variables to an instance of a class, we are forced to *re*-compile the class (or add a subclass) and to make the modification needlessly available to *all the class-instances*, thereby changing the whole program semantics (a waste of memory in case of addition of instance variables). Otherwise, we can add an *ad hoc* subclass in order to take into account the change of *in situ* behavior. If a class had a self-extension method, only the instances of the class which have dynamically executed this method would change their semantics (or allocate new memory) without the need of any re-compilation;

• plusieurs déclarations de sous-classes pourraient être épargnées si, dans leurs classes parentes, on déclarait une méthode d'auto-extension ;

• many sub-class declarations could be easily explained away if suitable self-extension methods in the parent class were available;

• les *coercions* vers le bas pourraient être implantées facilement à travers des méthodes d'auto-extension ;
• une sémantique formelle des méthodes de classe `addinstancevariable` et `addclassvariable` de Smalltalk pourrait être donnée à travers des méthodes d'auto-extension ;
• le processus d'*apprentissage* pourrait être facilement modélisé en utilisant un objet qui réagit au «message de l'enseignant» en lui donnant la possibilité d'accomplir, dans le futur, une nouvelle tâche en réponse à une requête de l'environnement extérieur (un vieux chien pourrait sembler apprendre de nouveaux tours si, dans sa jeunesse, on lui avait enseigné un tour «d'auto-apprentissage») ;
• le processus de *vaccination* contre le virus $\mathcal{X}$ pourrait être vu comme la capacité du système immunitaire de produire, dans le futur, un nouveau type de «$\mathcal{X}$-anticorps» après la réception d'un message indiquant qu'une $\mathcal{X}$- infection est en cours.

Cet aspect dynamique est typique des langages objets purs, *à la* Self; dans le modèle objets purs, les objets sont créés à partir des objets déjà existants, qui sont utilisés comme des prototypes. Les langages objets purs permettent naturellement la reclassification. Bien que Peter soit un langage à classes pur, dans certains cas, son caractère très statique lui pèse et semble le rendre inadapté à des applications qui demandent un haut niveau de dynamisme. Peter sait bien qu'il pourrait devenir plus dynamique s'il changeait son système de types statique pour un système de types dynamiques *à la* Smalltalk ; par contre, Peter croit fortement dans les types et dans tous les avantages d'être un langage typé statiquement.

Pour aider Peter dans son dilemme existentiel, on lui a conçu une extension, appelée Fly Peter, où les objets peuvent *s'échapper de leur propre classe* lorsqu'ils reçoivent un message. Cette extension demande très peu de modifications dans la syntaxe, dans la sémantique opérationnelle et dans le système de types, appelé $\vdash_{\mathsf{fly}}$. On commence avec deux «classes classiques» :

• *down-casting* could be smoothly implementable on objects with self-extension methods.
• an alternative principled explanation of the Smalltalk class methods `addinstancevariable` and `addclassvariable` could be given naturally using self-extension;
• the process of *learning* could be easily modeled using an object which can react to the "teacher's message" by extending its capability of performing, in the future, a new task in response to a new request from the environment (an old dog could appear to learn new tricks if in his youth he had been taught a "self-extension" trick);

• the process of *vaccination* against the virus $\mathcal{X}$ can be viewed as the act of extending the capability of the immune system of producing, in the future, a new kind of "$\mathcal{X}$-antibodies" upon receiving the message that an $\mathcal{X}$-infection is in progress.

Such a dynamic behavior is peculiar to object-based languages, *à la* Self, where objects are created from existing objects used as prototypes, and where inheritance occurs at the object level. Object-based languages naturally feature object reclassification. Although Peter is a pure class-based language, in some cases it is too *static* and not suited to applications featuring a high level of object-based dynamicity. Peter knowns that it could move towards dynamically-typed languages *à la* Smalltalk ; however, Peter strongly believes in types and in the advantages of having a statically-typed discipline.

To help Peter in its existential dilemma, we design an extension of Peter, called Fly Peter, where objects *can escape from their classes* upon a reception of a message. This extension requires few modifications in syntax, in operational semantics and in the type system, called $\vdash_{\mathsf{fly}}$. We start with the "classic two classes":

```
class Point extends Object
{int x;int y;
 Point(int x,int y){super();this.x = x;this.y = y;}
 Point move(int x,int y){return new Point(thix.x + x,this.y + y);}}

class Pix_Point extends Point
{bool pix;
 Pix_Point(int x,int y,bool pix){super(x,y);this.pix = pix;}
 bool onoff(){return not(this.pix);}}
```

Table 4.1: Encore les classes `Point` et `Pix_Point` / Still `Point` and `Pix_Point` classes

...et on présente tout de suite un exemple de l'extension qui donnera à Peter la *liberté de voler*...

...and we present quickly one extension that bestows the *freedom to fly* on Peter...

```
class Sharp_Point_1 extends Object
{int x;int y;
 Sharp_Point_1(int x,int y){super();this.x = x;this.y = y;}
 Sharp_Point_1 move(int x,int y){return new Sharp_Point_1(thix.x+x,this.y+y);}}
 bool onoff_pix(bool pix){modify {bool pix;
                                  bool onoff(){return not(this.pix);}}
                             and {return this.onoff();}}

Point        p = new Point(0,0);
Pix_Point    q = new Pix_Point(1,1,true);
Sharp_Point_1 r = new Sharp_Point(1,1)
...
if   p.x == q.x
then q.onoff()
else r.onoff_pix(true).onoff()       add_pix adds pix,onoff() to r and calls onoff()
     r is finally e-s-c-a-p-e-d from Sharp_Point_1 and belong to an anonymous subclass of it!
```

Table 4.2: La classe `Sharp_Point_1` dans Fly Peter / Fly Peter's `Sharp_Point_1` Class

La classe `Sharp_Point_1` a un nouveau type de méthode `onoff_pix` qui modifie l'objet qui le reçoit. Intuitivement, l'expression `modify` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}\}$ `and` $\{$`return e;`$\}$ définit une sous-classe de `Sharp_Point_1` (anonyme), avec les nouvelles variables et méthodes $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}\}$, puis crée un objet de cette nouvelle classe et, enfin, laisse cet objet exécuter l'expression qui se trouve après le `and`. Dans la table en haut, une variable `pix` et une nouvelle méthode `onoff` seront ajoutées à l'instance `r` de `Sharp_Point_1`.

The class `Sharp_Point_1` has a new kind of self-extending method `onoff_pix` which modifies the object that receives it. Intuitively, the expression `modify` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}\}$ `and` $\{$`return e;`$\}$ first it creates a new (anonymous) sub-class of `Sharp_Point_1`, with new variables and methods $\{\overline{\texttt{C}}\,\overline{\texttt{f}};\overline{\texttt{M}}\}$, then it creates a new object of this subclass. Finally, it executes the expression after the `and` on the new created object. In the above table, a `pix` variable and a new `onoff` method are added to the object instance `r` of `Sharp_Point_1`.

4.1.1 Syntaxe / Syntax

On modifie la syntaxe avec des méthodes qui peuvent modifier l'objet lui-même. L'ensemble $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}\}$ liste les variables et les méthodes à rajouter dans la sous-classe anonyme de la classe qui définit la méthode auto-modifiante ; `e` est l'expression à évaluer dans l'instance de la sous-classe anonyme ; les paramètres $(\overline{\texttt{D}}\ \overline{\texttt{x}})$ prennent en compte l'initialisation des variables d'instance $\overline{\texttt{C}}\ \overline{\texttt{f}}$ ainsi que les paramètres formels de l'expression `e`.

The only extension in the syntax lies in the possibility to modify the object itself. The set $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}\}$ lists variables and methods to be added in the anonymous subclass of the class defining the self-extending method; `e` is the expression to be evaluated in the new instance of the anonymous subclass; formal parameters $(\overline{\texttt{D}}\ \overline{\texttt{x}})$ take into account the initialization of the new instance variables $\overline{\texttt{C}}\ \overline{\texttt{f}}$ and the formal parameters of expression `e`.

`M`	::= `C m(`$\overline{\texttt{D}}\ \overline{\texttt{x}}$`){modify {`$\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}$`} and {return e;}}`	Self Modifying Methods

Table 4.3: Syntaxe de Fly Peter avec les méthodes auto-modifiantes / Fly Peter's Syntax with self-modifying methods

4.1.2 Exemples / Examples

Exemple : élimination de sous-classes / Example: Avoiding Subclass Declarations

L'exemple suivant montre comment factoriser un certain nombre de sous-classes dans Fly Peter.

The next example shows how to factorize subclasses in Fly Peter.

```
class A extends Object                                          write any class A
{...;...}

class B extends A                                               write any subclass B of A
{...;...
 spinach(){...}
}

class Sharp_A                  classes A and B can be collapsed into one sharp class Sharp_A
 {...;...
  popeye(){modify {Object spinach(){...}}
              and {return this.spinach();}}}                          self-extending method

(new Sharp_A()).popeye()                                  popeye add and call spinach
```

Table 4.4: La classe de `Bras de Fer` dans Fly Peter / Fly Peter's `Popeye` class

Exemple: redéfinition d'une méthode par elle-même / Example: Self-override of a Method

Il est bien connu qu'une sous-classe est utile non seulement pour ajouter de nouvelles variables d'instance ou de nouvelles méthodes mais aussi pour redéfinir des méthodes existantes.

It is well known that a subclass can be useful not only to add new instances of new methods but also to override existing methods. Self-modifying methods can modify other methods

Les méthodes auto-modifiantes peuvent faire cela facilement.

in an easy way.

```
class Sharp_Point_2 extends Object
{int x;int y;
 Sharp_Point_2(int x,int y){super();this.x = x;this.y = y;}
 Sharp_Point_2 move(int x,int y){return new Sharp_Point_2(thix.x+x,this.y+y);}
 Sharp_Point_2 change_move(int x,int y){modify {Sharp_Point_2 move(int x,int y)
                                  {return new Sharp_Point_2(this.x*x,this.y*y);}}
                                          and {return this.move(x,y);}}}

Sharp_Point_2 p = new Sharp_Point_2(100,100);              p is a point with x,y=100
              p.move(100,100);                          return a new point with x,y=200
              p.change_move(100,100);                 return a new point with x,y=10000
              p.change_move(10,10).move(20,20)  still return a new point with x,y=20000
```

Table 4.5: Une classe de Fly Peter avec une méthode qui redéfinit une méthode déjà existante / Fly Peter's Class with a Method that Override an Already Existing Method

Exemple : auto-redéfinition de la méthode soi-même / Example: Self-override of the Method Itself

Les méthodes auto-modifiantes peuvent aussi se modifier elle-mêmes : cela peut sembler amusant mais, au final, pas vraiment utile, car à chaque appel la méthode sera redéfinie mais utilisera toujours le même corps.

Self-modifying methods can even modify the method itself; this looks funny, and actually is not to very useful, since at every method call, the method is always overridden with the same body.

```
class Sharp_Point_3 extends Object
{int x;int y;
 Sharp_Point_3(int x,int y){super();this.x = x;this.y = y;}
 int get_install(){modify{int get_install(){return this.x;}}
                      and{return this.get_install();}}}

Sharp_Point_3 p = new Sharp_Point_3(100,100);              p is a point with x,y=100
              p.get_install();                                              return 100
              p.get_install().get_install();                          still return 100
```

Table 4.6: Une classe qui se modifie elle-même dans Fly Peter/ Fly Peter's Class that Self-modify in Itself

Exemple : sauver de la mémoire / Example: Save Memory Space

L'exemple suivant montre comment sauver de la mémoire lorsqu'on veut créer un grand nombre d'instances (les chèvres blanches) et une instance «spéciale» (la chèvre noire) qui a des variables supplémentaires.

The next example shows how to save memory space when we plan to instantiate a huge number of object instances (the white goats), and one "special" instance (the black goat) that has one extra variable.

```
class Sharp_Goats extends Object
{int x;
 Sharp_Goats(int x){super();this.x = x}
 black_goat(String s){modify {String feature = s;} and {return this.feature;}}}

for (int i = 1; i = 666) a little abuse of notation in Peter: the for expression is derivable
 {Sharp_Goats white_goat_i = new Sharp_Goat(i);}    create 666 Sharp_Goats instances
...                                                 manipulate 666 Sharp_Goats instances
white_goat_666.black_goat("I can speak!").feature   an extra slot is added only for
  the 666th demonic object white_goat_666 that does not belong anymore to class Sharp_Goat
```

Table 4.7: Un exemple «d'enfer» dans Fly Peter/ Fly Peter's "Demonic" Example

Exemple : exceptions *ad hoc* avec terminaison / Example: *ad hoc* Exception Handling with Termination

L'exemple suivant montre comment définir une exception propre à une classe `Sharp_B` qui sera déclarée et exécutée seulement si l'exception est vraiment soulevée. La méthode `my_try` exécute du code protégé ; si une exception est soulevée via la méthode `my_throws(error code)`, alors la classe `Sharp_B` s'auto-modifiera avec l'ajout d'une méthode `my_catch(int ec)` et son exécution immédiate, afin de traiter l'exception *in situ*.

This example defines one exception private to class `Sharp_B`: the exception is declared and executed only if really arised. The method `my_try` wraps protected code; if one exception is raised, via method `my_throws(error code)`, then the class `Sharp_B` is self-inflicted with the method handler `my_catch(int ec)`, and finally the handler is executed *in situ*.

```
class Sharp_B extends Objects
{...;...
 String my_try(){return ... if test then this.my_throws(error code)
                                    else "All OK"}
 String my_throws(int code){modify {String my_catch(int ec){...handle except...}}
                              and {return this.my_catch(code);}}}

Sharp_B b = new Sharp_B();
  b.my_try()                                                    b calls my_try
  -> b.my_throws(0)          my_try extends b with my_catch and call it with error code 0
     -> b.my_catch(0)           b is escaped from Sharp_B: it feature an exception handler
```

Table 4.8: Une classe avec traitement des exceptions *ad hoc* en Fly Peter / Fly Peter's Class with an *ad hoc* Exception Handling Mechanism

Exemple : exceptions avec reprise / Example: Exception Handling with Resumption

L'exemple est pratiquement similaire au précédent, à part le fait qu'une fois que l'exécution a été traitée, on réessaye d'exécuter la méthode protégée. On appelle ce mécanisme «exceptions avec reprise».

This example behaves essentially as the previous one: the only difference is that we retry the protected body, after the exception handler has been executed. We call this mechanism "exception handling with resumption".

```
class Sharp_C extends Object
{...;...
 String my_try(){return ... if test then this.my_throws(error code)
                                    else "All OK"}
 String my_throws(int code){modify {String my_catch(int ec){...handle except...}}
                               and {return this.my_catch(code).my_try();}}}

Sharp_C c = new Sharp_C();
  c.my_try()                                                        c calls my_try
  -> c.my_throws(0)          my_try extends c with my_catch and call it with error code 0
     -> c.my_catch(0).my_try()      moral: the exception is handled locally and we re-try...
```

Table 4.9: Une classe avec exceptions avec reprise en Fly Peter / Fly Peter's Class with an Exception Handling Mechanism with Resumption

Exemple : coercions vers le bas et sous-typage structurel / Example: Downcasting and Nominal Subtyping

L'exemple suivant n'est pas typable actuellement dans le système de types de Fly Peter à cause du manque de typage dit *structurel* ; le code de l'exemple est bloqué par le typage mais il pourrait s'exécuter sans avoir d'erreurs à l'exécution. Pour typer cet exemple il faudrait mélanger sous-typage nominal et structurel subtilement, ce qui sera sûrement proposé dans Fly Peter V2.

The next example actually is not typable in the Fly Peter type system because of lack of *nominal* subtyping: the example's code is not blocked at compile time, but it could be executed without run-time errors. In order to type this example, we would smoothly mix nominal and structural subtyping: this will surely be proposed in Fly Peter V2.

```
class Pix_Point extends Point
{bool pix;
 Pix_Point(int x,int y,bool pix){super(x,y);this.pix = pix;}
 bool equal(Pix_Point p){return this.x==p.x && this.y==p.y && this.pix==p.pix;}}

class Sharp_Point_4 extends Object
{int x;int y;
 Sharp_Point_4(int x,int y){super();this.x = x;this.y = y;}
 bool equal_pix(bool pix){modify {bool pix;
                                  bool equal(Pix_Point p){return this.x==p.x &&
                                                              this.y==p.y && this.pix==p.pix;}}}
                             and {return this.onoff();}}

   Pix_Point p = new Pix_Point(1,1,true);
Sharp_Point_4 q = new Sharp_Point_4(1,1)
Sharp_Point_4 r = new Sharp_Point_4(1,1)
(1).. p.equal(q/r)                                                    not typable
(2).. q/r.equal_pix(true).equal(q/r.equal_pix(true))                      typable
(3).. q/r.equal_pix(true).equal(p)  typable only mixing nominal and structural subtyping
(4).. p.equal(q/r.equal_pix(true))  typable only mixing nominal and structural subtyping
```

Table 4.10: Une aperçu sur comment mélanger sous-typage nominal et structurel dans Fly Peter/ Spot About how Combine Nominal and Structural Subtyping in Fly Peter

Un petit commentaire sur les appels de la méthode `equal` est nécessaire :
• `(1)` visiblement `Pix_Point` et `Sharp_Point_4` ne sont pas en relation de sous-typage ;
• `(2)` l'exécution de la méthode auto-modifiante `equal_pix` ajoute `pix,equal` à `q,r`, donc l'expression est typable ;
• `(3)` l'exécution de la méthode auto-modifiante `equal_pix` ajoute `pix,equal` à `q,r`, qui se sont échappés de la classe `Sharp_Point_4` ; ils appartiennent à une sous-classe anonyme de `Sharp_Point_4` qui s'avère être aussi une sous-classe *structurelle* de `Pix_Point` ; vu que le sous-typage en Peter est pour l'instant nominale, cette expression n'est pas typable même si, à l'exécution, elle ne produirait pas d'erreurs ;
• `(4)` Même raisonnement que dans le point `(3)`.

A short comment about the three calls of the `equal` method is needed:
• `(1)` clearly `Pix_Point` and `Sharp_Point_4` are incomparable;
• `(2)` the execution of the self-extending method `equal_pix` adds `pix,equal` to `q,r`, so the expression is typable;
• `(3)` the execution of the self-extending method `equal_pix` adds `pix,equal` to `q,r`, allowing them to escape from the `Sharp_Point_4` class; they now belong to a direct anonymous subclass of `Sharp_Point_4` that is also a subclass of `Pix_Point` using *structural* subtyping; since the actual Peter's subtyping mechanism is nominal, this expression is not typable even if it will not crash at run-time;
• `(4)` For the same reasons as above in `(3)`, the expression is not typable.

4.1.3 Règle clé pour l'exécution / Key Run-time Rule

La règle principale de la sémantique opérationnelle de Fly Peter est celle qui auto-modifie un objet lorsqu'il reçoit un message.

The principal rule of the operational semantics of Fly Peter is the one that self-modifies an object upon reception of a message.

$$\frac{mbody(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{E},\mathtt{modify}\ \{\overline{\mathtt{A}}\ \overline{\mathtt{a}};\overline{\mathtt{N}}\}\ \mathtt{and}\ \{\mathtt{return}\ \mathtt{e}_0;\})}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{f}},\overline{\mathtt{g}}) \longrightarrow [\overline{\mathtt{g}}/\overline{\mathtt{x}},\mathtt{new}\ \mathtt{E}(\overline{\mathtt{e}},\overline{\mathtt{f}})/\mathtt{this}]\mathtt{e}_0}\ (\text{Run·Modify})$$

Table 4.11: Règle clé pour l'exécution / Key Run-time Rule

Dans une «coquille de noix» :
• On cherche, à travers la fonction *mbody*, la méthode auto-modifiante `m` ;
• On accède à la base de données des classes `CL` pour avoir une définition complète de la classe `C` ;
• On crée une nouvelle sous-classe anonyme `E` de `C` qui contient toutes les variables d'instances et les nouvelles méthodes définies dans la méthode auto-modifiante ;
• On exécute le corps $\mathtt{e}_0$ en substituant les paramètres formels et la méta-variable `this` avec une nouvelle instance de `E`.

In a nutshell:
• We fetch, via the *mbody* function, the self-modifying method `m`;
• We look in the class table `CL` for the complete definition of class `C`;
• We create a fresh subclass `E` of `C` containing all the new instance variables and the new methods defined in the self-extended method;
• We call the body $\mathtt{e}_0$ by instantiating the formal parameters and the metavariable `this` with a new instance of `E`.

4.1.4 Règles clés pour le typage / Key Type-checking Rules

Avant de présenter les règles les plus importants du système de types, appelé $\vdash_{\mathsf{fly}}$, on mon-

Before presenting the type rules, called $\vdash_{\mathsf{fly}}$, let's show how to type-check simple expressions ma-

tre comment typer des expressions très simples qui manipulent `Sharp_Point_1` :

nipulating `Sharp_Point_1`:

```
p = new Sharp_Point_1(0,0);                                   instance p of Sharp_Point_1
p : Sharp_Point_1                                             will belongs to Sharp_Point_1
p.onoff_pix(true);                                            we call add_pix and its type changes in
p.onoff_pix(true) : Anonymous_#666          this new class is a subclass of Sharp_Point_1
p.onoff_pix(true).onoff()          created on the ''fly'': the onoff() method will respond
p.onoff_pix(true).onoff() : bool typable thanks to the definition of class Anonymous_#666
```

Table 4.12: Exemples de typage dans Fly Peter/ Fly Peter's Typing Examples

où la classe anonyme est définie ainsi :

where the anonymous class is defined as:

```
class Anonymous_#666 extends Sharp_Point_1
{bool pix;
 Anonymous_#666(int x,int y,bool b){super(x,y);this.pix = b;}
 bool onoff(){return not(this.pix);}}
```

Table 4.13: Une classe anonyme dans Fly Peter / Fly Peter's Anonymous Class

Après le nombre important d'exemples qu'on vient de présenter dans ce chapitre, le raisonnement de Fly Peter est que chaque objet qui reçoit un message relatif à une méthode auto-modifiante peut s'échapper de sa propre classe et *a fortiori* voir son type changer. Cependant, cette fuite a été bien conçue à travers la définition d'une méthode auto-modifiante : sa nouvelle règle de typage est :

After the quite large number of examples, the rationale of Fly Peter is that upon the reception of a self-modifying message, the object escapes from its class and *a fortiori* its type changes. Nevertheless, this escape has been learned *in his youth*, *i.e.* the method we are calling must be declared as a self-modifying method whose typing rule is:

$$\frac{\begin{array}{l}\texttt{CL}(\texttt{C}) = \texttt{class C extends D } \{...; \texttt{C}(\overline{\texttt{A}}\ \overline{\texttt{a}})\{...\}\ ...\} \\ \texttt{E fresh} \qquad \texttt{K}' = \texttt{E}(\overline{\texttt{A}}\ \overline{\texttt{a}}, \overline{\texttt{B}}\ \overline{\texttt{b}})\{\texttt{super}(\overline{\texttt{a}}); \texttt{this}.\overline{\texttt{b}} = \overline{\texttt{b}};\} \\ \texttt{class E extends C } \{\overline{\texttt{B}}\ \overline{\texttt{b}}; \texttt{K}'\ \overline{\texttt{N}}\}\ \texttt{OK} \qquad \text{update CL with E} \\ \overline{\texttt{y}}{:}\overline{\texttt{C}}, \texttt{this:E} \vdash_{\mathsf{fly}} \texttt{e} : \texttt{G} \qquad \textit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \to \texttt{G}) \qquad \texttt{F <: G}\end{array}}{\texttt{F m}(\overline{\texttt{B}}\ \overline{\texttt{x}}, \overline{\texttt{C}}\ \overline{\texttt{y}})\{\texttt{modify } \{\overline{\texttt{B}}\ \overline{\texttt{b}}; \overline{\texttt{N}}\} \texttt{ and } \{\texttt{return e};\}\}\ \texttt{OK IN C}}\ (\text{Meth·Ok·Class·Fly})$$

Table 4.14: La nouvelle règle de typage pour l'appel de méthode en Fly Peter / Fly Peter's New Type Rule for Method Call

Dans une «coquille de noix» :
• On récupère la définition de la classe `C` et de

In a nutshell:
• We fetch the definition of class `C` with its con-

son constructeur $\texttt{C}(\overline{\texttt{A}}\ \overline{\texttt{a}})\{...\}$;
• On crée une nouvelle sous-classe anonyme `E` de `C` qui contient toutes les variables d'instance et les nouvelles méthodes définies dans la méthode auto-modifiante ; le constructeur $\texttt{K}'$ est construit à partir du constructeur de `C` ;
• On met à jour la base de données des classes `CL` avec la nouvelle classe anonyme `E` ;
• On type le corps `e` de la méthode `m` dans un contexte enrichi des types des paramètres actuels et de la méta-variable `this` associée à `E`. Le reste de la règle se comporte comme dans (Meth·Ok·Class) de Baby Peter.

structor $\texttt{C}(\overline{\texttt{A}}\ \overline{\texttt{a}})\{...\}$;
• We create a fresh subclass `E` of `C` containing all the new instance variables and the new methods defined in the self-extended method; the constructor $\texttt{K}'$ is built over the `C`'s constructor;
• We update the database of all classes `CL` with the new anonymous class `E`;
• We type-check the body `e` of method `m` in a context enriched with the types of the actual parameters and the type of the metavariable `this` assigned to `E`. The rest of the rule behaves as in the original (Meth·Ok·Class) rule of Baby Peter.

Sources d'inspiration / Inspiration sources

Ce chapitre a été inspiré par les deux articles sur les langages objets purs les plus intrigants que j'ai écrits [35] et [37] avec Pietro et Furio. D'autres lectures sur la re-classification en langages *à la* Java sont contenues dans la «saga» de Fickle, inspirée par [37], de Sophia, Ferruccio, Mariangiola et Paola.

This chapter has been influenced by the two most beautiful papers on object-based languages I have ever written [35], and [37] with Pietro and Furio. Other reading on object reclassification in Java-like languages are the Sophia, Ferruccio, Mariangiola, and Paola's Fickle saga, [37]'s inspired, [DDDCG01, DDDCG02].

Part III

Le vieux Peter (raisonnement) / Peter, the Old Man (Reasoning)

Chapter 5

Peter et l'île qui n'existe pas / Peter and the Nowhere Island

Ce chapitre est dédié à Claude et à l'Équipe Protheo de Nancy, en particulier à Horatiu, pour m'avoir infecté avec le «virus intellectuel» qui m'a fait mélanger les systèmes de réécriture avec toutes sortes de mécanismes de calcul...

This chapter is dedicated to Claude, and the Protheo Team in Nancy, especially Horatiu, for infecting me with the "intellectual virus" of blending term rewriting with any kind of computational mechanism...

5.1 L'île formelle de Peter / The Peter's Formal Island

On présente une extension de Peter, appelée Philo Peter qui offre la possibilité de définir le corps des méthodes d'une façon déclarative à travers des «*îles formelles*». Une île formelle est une méthode d'une classe Peter décrite d'une façon *déclarative*, c-à-d. à travers des règles de réécriture. Chaque règle de réécriture est appelée, dans le jargon de Peter, une *pæninsula* ; l'ordre des pæninsulæ n'est pas important. «Débarquer» sur une île de Philo Peter, à travers un appel de message, est une expérience enthousiasmante pour un Philo Peter-programmeur ; selon le paradigme à base de règles, les motifs des règles de réécriture, appelés *cape des pæninsulæ*, sont filtrés avec leurs arguments ; si plus d'un cape filtre les paramètres actuels de la méthode appelée, alors plus d'un résultat sera rendu à la sortie ; cela signifie qu'une exécution d'une île formelle peut produire des résultats multiples.

We present a Peter's extension, called Philo Peter that allows defining method bodies in a declarative way, using *"formal islands"*. A formal island is a Philo Peter class-method with a behavior described in a *declarative way*, *i.e.* through a formal set of rewrite rules. Every rewrite rule is called, in Philo Peter's jargon, a *pæninsula*; the order of pæninsulæ is not important. "Disembarking" on a Peter's island, via message passing, is an amazing experience for the Philo Peter-programmer; following the rule-based paradigm, all input patterns, pictorially called *pæninsulæ's cape*, are matched against the actual pattern; if more than one cape pass the filter, then more than one cape will be fired; this means that a run of a formal island can lead to multiple results.

5.1.1 Syntaxe / Syntax

La syntaxe des îles est la suivante :

The island syntax is the following:

`M`	::=	$\overline{\texttt{C}}$ `m(E,`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`,`$\overline{\texttt{D}}$ $\overline{\texttt{alg}}$`){return` $\overline{\texttt{e}}$`;}`	Methods containing a formal island
`e`	::=	`e -> e` | `alg(`$\overline{\texttt{e}}$`)`	Island′s idioms

Table 5.1: Syntaxe de Philo Peter pour des îles / Philo Peter's Syntax for Islands

Les méthodes d'une classe peuvent contenir des îles déclaratives ; `E` est le type de l'argument d'entrée, `C` est le vecteur des types des résultats possibles en sortie ; $\overline{\texttt{C}}$ $\overline{\texttt{x}}$ est un contexte de type contenant les types des variables libres des cape des pæninsulæ de la méthode. Les motifs sont des termes algébriques de la forme `alg(`$\overline{\texttt{e}}$`)`, où `alg` est une constante-algébrique typée (c-à-d. une formule atomique, des chaînes de caractères, des entiers, etc.). Dans un corps de pæninsula, l'utilisation de la méta-variable `this` permettra un appel récursif à la même ou à une autre île (ou méthode) de la même classe.

Class methods can contain declarative islands; `E` is the type of the actual argument provided as input, and $\overline{\texttt{C}}$ is the type-vector of the (possible multiple) output results; $\overline{\texttt{C}}$ $\overline{\texttt{x}}$ is the type-context containing the type of all the free-variables of the formal patterns (*i.e.* the cape) provided in the body of the island; $\overline{\texttt{D}}$ $\overline{\texttt{alg}}$ is another type-context containing types of algebraic-constants. Patterns are algebraic terms of the shape `alg(`$\overline{\texttt{e}}$`)`, where `alg` is a typed algebraic-constant (*e.g.* atomic formulas, strings, integers, etc.). Inside the body of a pæninsula, the metavariable `this` will allow recursive calls to the same or to an other pæninsula (or method) of the same class.

5.1.2 Exemples / Examples

Le premier exemple présente une île formelle qui calcule la fonction factorielle ; le second cape attache une *garde* au paramètre formel ; une garde est un test qui peut bloquer l'évaluation d'un cap ; le deuxième cape sera exécuté si le paramètre formel est différent de `1` : les gardes permettent de définir des cape mutuellement exclusifs. Le type du résultat est une liste qui contient les types des résultats des évaluations de toutes les pæninsulæ. Dans ce qui suit, on écrira $\texttt{C}^{\texttt{n}}$ pour $\underbrace{\texttt{C},\ldots,\texttt{C}}_{n \text{ times}}$.

The first example is a simple formal island computing the factorial function; in the second cap, a *guard* is attached to the formal parameter `n`. A guard is a condition that can block the firing of a cap; the second cape can be fired for any actual parameter different than `1`: guards simplify mutually exclusive cape definitions by allowing to fire at most one cape at the time. The return-type is the list containing the result-types of all pæninsulæ. In what follows, we write $\texttt{C}^{\texttt{n}}$ for $\underbrace{\texttt{C},\ldots,\texttt{C}}_{n \text{ times}}$.

```
class Nowhere_Island extends Object
{;super();
int² island(int,int n){return fact(1) -> 1,
                              fact(n\1) -> n*this.island(fact(n-1));}}

ile = new Nowhere_Island();
->ile.island(fact(4));                                          reduces to
  ->4*ile.island(fact(3))                                       reduces to
    ->3*ile.island(fact(2))                                     reduces to
      ->2*ile.island(fact(1))                                   reduces to
        ->1                                                base recursion case
->24                                   the final result is of type int supertype of int²
```

Table 5.2: La première île (factorielle) en Philo Peter/ The First Philo Peter's Island (Factorial)

La deuxième île transforme une formule logique en sa *forme normale négative* ; cette routine est utilisée pour implanter des procédures de décision dans les *prouveurs de modèles*. L'entrée est une formule de la logique propositionnelle avec la syntaxe suivante :

The second island computes a *negation normal form*, which is used in implementing *decision procedures*, present in almost all model checkers. The processed input is an implication-free language of logical formulas generated by the grammar:

```
alg ::= alg | and(alg,alg) | or(alg,alg) | not(alg)

class Logic_Island extends Object
{;super();
 bool[6] nnf(bool,bool x,bool y){return
                                alg1 -> alg1,
                                alg2 -> alg2,
                          not(not(x)) -> this.nnf(x),
                         not(or(x,y)) -> and(not(this.nnf(x)),not(this.nnf(y))),
                        not(and(x,y)) ->  or(not(this.nnf(x)),not(this.nnf(y))),
                            and(x,y) -> and(this.nnf(x),this.nnf(y)),
                             or(x,y) ->  or(this.nnf(x),this.nnf(y));}}
logic = new Logic_Island();
logic.nnf(and(not(and(alg1,alg2)),not(and(alg1,alg2)))); the formal island is called ...
-> and(or(not(alg1),not(alg2)),or(not(alg1),not(alg2))) of type bool supertype of bool[6]
```

Table 5.3: L'île de la forme normale négative / The Negation Normal Form's Island

La troisième et la quatrième îles montrent comment le filtrage de Philo Peter peut être non exclusif, c-à-d. plus d'une pæninsula peut être exécutée simultanément. De plus, le quatrième exemple contient un appel récursif.

The third and forth islands show one important point, namely that pattern matching needs not to be exclusive, *i.e.* more than one pæninsula can be simultaneously fired. Moreover, the forth example contains a recursive call.

```
class Non_Exclusive_Island extends Object
{;super();
 string[2] deduction(string,string name){return
                                        man(name)      -> mortal(name),
                                        man("Socrates")-> mortal("Socrates");}}

philo = new Non_Exclusive_Island();
philo.deduction(man("Socrates"))                          the formal island is called ...
-> (mortal("Socrates"),mortal("Socrates"))       ... and the result is of type string[2]

class Non_Exclusive_Rec_Island extends Object
{;super();
 string[3] deduction(string){return "Socrates" -> this.deduction("man"),
                                    "man"      -> "mortal",
                                    "Socrates" -> "mortal";}}

philo = new Non_Exclusive_Rec_Island();
philo.deduction("Socrates")                               the formal island is called ...
-> (mortal,mortal)            ... and the result is of type string[2], supertype of string[3]
```

Table 5.4: L'île non exclusive et l'île récursive / The Non-Exclusive and the Recursive Islands

La cinquième île montre qu'exécuter plusieurs pæninsulæ simultanément peut conduire à des résultats multiples :

The fifth island shows that the simultaneous firing of many pæninsulæ leads to different output-types:

```
class VIP_Island extends Object
{;super();
 (string²,integer) deduction(string){return "Socrates" -> this.deduction("man"),
                                            "man"      -> "mortal with dracmæ=",
                                            "Socrates" -> 10000;}}

philo = new VIP_Island();
philo.deduction("Socrates")                                  the formal island is called ...
-> (mortal,10000) ... result is of type (string,integer), supertype of (string²,integer)
```

Table 5.5: L'île des VIP / The VIP's Island

Le dernier exemple présente une classe avec des îles qui s'appellent récursivement.

The last example shows a class containing some islands calling each other recursively.

```
class Texas_Island extends Objects
{;super();
int² plus(int,int n int m){return (0,n)         -> n,
                                  (succ(n),m)   -> succ(this.plus(n,m));}
int³ mult(int,int n int m){return (0,m)         -> 0,
                                  (succ(0),m)   -> m,
                                  (succ(n\0),m) -> this.plus(m,this.mult(n,m));}
int² pow(int,int n int m){return  (n,0)         -> succ(0),
                                  (n,succ(m))   -> this.mult(n,this.pow(n,m));}}

TX45 = new Texas_Island();
TX45.pow(succ(succ(0)),succ(succ(0)))                    the formal island is called ...
->TX45.mult(succ(succ(0)),TX45.pow(succ(0)))      ...islands mult and pow are called...
  -> succ(succ(succ(succ(0))))      ...and some recursive calls, 4 is returned of type int
```

Table 5.6: L'île calculatrice / The Calculator's Island

5.1.3 Comment ça marche ? / How Can Work?

On commence à lister les différences entre les îles de Philo Peter et la construction du genre «match-case» *à la* ML :
• en ML, les branches d'entrée sont mutuellement exclusives, tandis que les pæninsulæ de Philo Peter peuvent être exécutées simultanément ;
• en ML, les types des branches d'entrée peuvent être différents et un seul type de sortie est permis, tandis que dans Philo Peter, les types des branches d'entrée doivent être égaux, mais

We start by summarizing the differences between Philo Peter's formal islands and usual match-cases *à la* ML:
• in ML match-cases, input branches need to be mutually exclusive, while Philo Peter's pæninsulæ can be fired simultaneously;
• in ML match-cases, input branches types can be different but only a single output-type is allowed, while in Philo Peter's pæninsulæ, all input branches types must be equal, while

les types des sorties peuvent être différents.

Le raisonnement est résumé d'une façon ludique ci-dessous avec un petit peu d'anglais.

1 *i.e.* By means of a joke.

output-types can be different.

The rationale is "lunaparkly"[1] summarized below.

For types ♢, ♠, ♣, ♡, and ⊙:

- In ML-like match-case constructs, we have a behavior that looks-like a *Beretta*-gun!

in		out
♢	⟶	⊙
♠	⟶	⊙
♣	⟶	⊙
♡	⟶	⊙

- In Philo Peter's pæninsulæ, we have a behavior that looks-like a *Kalashnikov* machine gun!

in		out
⊙	⟶	♢
⊙	⟶	♠
⊙	⟶	♣
⊙	⟶	♡

Table 5.7: Un luna park pour mon filtrage / A Luna Park for My Matching

La clé du typage d'une île en Philo Peter est à chercher dans l'application d'un argument à toutes les pæninsulæ d'une île ; ceci peut produire plusieurs résultats. Le type du résultat contiendra les types des résultats obtenus par les pæninsulæ dont le cape a filtré l'argument en entrée. Ce type sera un *super-type* de tous les résultats possibles des pæninsulæ de l'île.

Comme exemple, si on applique une île avec deux pæninsulæ de type $\mathtt{C}_1 \to \mathtt{C}_2$ et $\mathtt{C}_1 \to \mathtt{C}_3$ à un argument de type $\mathtt{C}_1$, on obtient un résultat de type $(\mathtt{C}_2, \mathtt{C}_3)$. Pour aider le programmeur en Philo Peter à comprendre le comportement d'une île (qui est une conséquence directe de la distribution de l'application à toutes les pæninsulæ), je présente la fonction partielle *arr* sur les types, qui transforme le type d'une pæninsula en un type fonction :

The main point in Philo Peter's island type system lies in applying an island to an argument, producing a result obtained by dispatching the argument to all island's cape. The output-type will contain all the output-types of every pæninsula whose cape *matches* against the input argument. This type will be a *supertype* of all possible pæninsulæ's result-types in the island.

As a simple example, if we apply an island with two pæninsulæ of types $\mathtt{C}_1 \to \mathtt{C}_2$ and $\mathtt{C}_1 \to \mathtt{C}_3$ to an argument of type $\mathtt{C}_1$, then we obtain a result of type $(\mathtt{C}_2, \mathtt{C}_3)$. To help a Philo Peter-programmer to understand Philo Peter's island behavior (which is a direct consequence of dispatching application to all pæninsulæ), it is useful to present the following partial function *arr* on types, which transforms a pæninsulæ-type into a function-type:

$$
\begin{array}{lcll}
arr(\mathtt{C}_1 \to \mathtt{C}_2) & = & \mathtt{C}_1 \to \mathtt{C}_2 & \\
arr(\mathtt{C}_1, \mathtt{C}_2) & = & \mathtt{C}_3 \to (\mathtt{C}_4, \mathtt{C}_5) & \left\{ \begin{array}{ll} \text{if} & arr(\mathtt{C}_1) = \mathtt{C}_3 \to \mathtt{C}_4 \\ \text{and} & arr(\mathtt{C}_2) = \mathtt{C}_3 \to \mathtt{C}_5 \end{array} \right.
\end{array}
$$

Table 5.8: La fonction qui normalise les types des îles / The Island's Normalizing Function

5.1.4 Règles clés pour l'exécution / Key Run-time Rules

Les règles clés à retenir sont celles qui appellent une île et celles qui exécutent l'île même.

The main rules to keep in mind are the one invoking an island and the one performing in parallel computations in pæninsulæ.

$$\frac{mbody(\mathtt{m},\mathtt{C}) = (\mathtt{island}, \overline{\mathtt{e_i}\ \mathtt{->}\ \mathtt{e'_i}}) \quad \overline{\theta}_\mathtt{j} = \overline{\mathsf{filter}}(\overline{\mathtt{e}}_\mathtt{i}, \mathtt{e}_0)}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{m}(\mathtt{e}_0) \longrightarrow [\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})/\mathtt{this}]\overline{\mathtt{e_j}\theta_\mathtt{j}}}\ (\text{Island·Call})$$

$$\frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_3 \quad \mathtt{e}_2 \longrightarrow \mathtt{e}_4}{\mathtt{e}_1\,,\mathtt{e}_2 \longrightarrow \mathtt{e}_3\,,\mathtt{e}_4}\ (\text{Island·Par})$$

Table 5.9: Règles clés pour l'exécution des îles / Key Run-time Rules for Islands

Dans une «coquille de noix» :
• (Island·Call) Dans cette règle, tous les cape des îles sont filtrés avec le paramètre actuel d'entrée : une substitution est produite ; tous les cape qui filtrent les paramètres actuels produisent l'évaluation de leur pæninsula ; les cape qui ne filtrent pas laissent leur pæninsula non évaluée : plus d'un résultat peut être produit ;
• (Island·Par) Cette règle est une règle classique de congruence.

In a nutshell:
• (Island·Call) In this rule, all island's cape are matched against the actual parameters: a substitution is produced; all cape that pass the pattern fire their pæninsulæ *in parallel*; all cape that do not pass the filter discard their pæninsulæ; then, the evaluation continues in every pæninsula independently; more than one result can be produced;
• (Island·Par) This rule is a classical congruence rule.

5.1.5 Règles clés pour le typage / Key Type-checking Rules

Pour typer les îles de Philo Peter il ne faut pas modifier ou ajouter un grand nombre des règles ; on les présente comme suit :

Few typing rules are added and modified in order to type-checks Philo Peter's island. The most important are presented below:

$$\frac{\Gamma \vdash_{\mathsf{philo}} \mathtt{e}_0 : \mathtt{C}_0 \quad mtype(\mathtt{m},\mathtt{C}_0) = \mathtt{D} \to \overline{\mathtt{E}},\overline{\mathtt{F}} \quad \Gamma \vdash_{\mathsf{philo}} \mathtt{e} : \mathtt{G} \quad \mathtt{G} <: \mathtt{D}}{\Gamma \vdash_{\mathsf{philo}} \mathtt{e}_0.\mathtt{m}(\mathtt{e}) : \overline{\mathtt{E}}}\ (\text{Type·Call·Island})$$

$$\frac{\Gamma \vdash_{\mathsf{philo}} \mathtt{e}_1 : \mathtt{C} \quad \Gamma \vdash_{\mathsf{philo}} \mathtt{e}_2 : \mathtt{D}}{\Gamma \vdash_{\mathsf{philo}} \mathtt{e}_1\ \mathtt{->}\ \mathtt{e}_2 : \mathtt{C} \to \mathtt{D}}\ (\text{Type·Arrow}) \qquad \frac{\Gamma \vdash_{\mathsf{philo}} \mathtt{e_i} : \mathtt{C_i} \quad \mathtt{e_i} \in \overline{\mathtt{e}} \quad \mathtt{C_i} \in \overline{\mathtt{C}}}{\Gamma \vdash_{\mathsf{philo}} \overline{\mathtt{e}} : \overline{\mathtt{C}}}\ (\text{Type·Set})$$

$$\frac{\Gamma \vdash_{\mathsf{philo}} \mathtt{alg} : \overline{\mathtt{C}} \to \mathtt{E} \quad \Gamma \vdash_{\mathsf{philo}} \overline{\mathtt{e}} : \overline{\mathtt{D}} \quad \overline{\mathtt{D}} <: \overline{\mathtt{C}}}{\Gamma \vdash_{\mathsf{philo}} \mathtt{alg}(\overline{\mathtt{e}}) : \mathtt{E}}\ (\text{Type·Alg})$$

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\ldots\} \quad override(\mathtt{m},\mathtt{D},\overline{\mathtt{C}} \to \overline{\mathtt{E}}) \\ \overline{\mathtt{x}}{:}\overline{\mathtt{C}}, \mathtt{this}{:}\mathtt{C} \vdash_{\mathsf{baby}} \mathtt{usetype}\ \{\mathtt{e_{ile}}\}\ \mathtt{in}\ \{\mathtt{e}_0\} : \overline{\mathtt{F}} \quad arr(\overline{\mathtt{F}}) = \mathtt{G} \to \overline{\mathtt{H}} \quad \overline{\mathtt{H}} <: \overline{\mathtt{E}}\end{array}}{\overline{\mathtt{E}}\ \mathtt{m}(\mathtt{G},\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e}_0;\}\ \mathtt{OK\ IN\ C}}\ (\text{Meth·Ok·Class·Island})$$

Table 5.10: Règles clés pour le typage des îles en Philo Peter / Philo Peter's Key Type-checking Rules for Islands

Dans une «coquille de noix» :
• (Type·Call·Island) Dans cette règle, le type de l'île est $\mathtt{D} \rightarrow \overline{\mathtt{E}}, \overline{\mathtt{F}}$, où $\overline{\mathtt{E}}, \overline{\mathtt{F}}$ représente le type du résultat de toutes les pæninsulæ ; le type G du paramètre actuel est un sous-type du type du paramètre formel D ; le type du résultat d'un appel à une île $\overline{\mathtt{E}}$ est un sous-type (en largeur) du type $\overline{\mathtt{E}}, \overline{\mathtt{F}}$ que l'on obtiendrait si toutes les pæninsulæ étaient exécutées. Cette relation de sous-typage est correcte car quelques cape ne filtreront pas avec le paramètre actuel à cause d'un échec de filtrage.
• (Type·Arrow) Cette règle est utilisée pour typer une règle d'un système de réécriture (a.k.a. une pæninsula) ; les variables libres des $\mathtt{e}_1$ et $\mathtt{e}_2$ sont déjà chargées dans le contexte Γ ;
• (Type·Set) Cette règle assigne un type-cartésien à une île ; ce type est composé de tous les types de pæninsulæ de l'île ;
• (Meth·Ok·Class·Island) Soit G le type du paramètre formel : en utilisant notre système de types *pluggable*, on type l'île, *i.e.* toutes les pæninsulæ, dans le système de types $\vdash_{\mathsf{philo}}$, codé par $\mathtt{e}_{\mathtt{ile}}$, en produisant le type $\overline{\mathtt{F}}$: on appelle la fonction *arr* sur $\overline{\mathtt{F}}$ pour obtenir le type fonctionnel de l'île $\mathtt{G} \rightarrow \overline{\mathtt{H}}$; finalement, on vérifie que le type du résultat de l'exécution de l'île $\overline{\mathtt{H}}$ est «point-à-point» un sous-type du type E déclaré comme type résultat de l'île.

In a nutshell:
• (Type·Call·Island) In this rule, the island signature is $\mathtt{D} \rightarrow \overline{\mathtt{E}}, \overline{\mathtt{F}}$, where $\overline{\mathtt{E}}, \overline{\mathtt{F}}$ denotes the result-type of all pæninsulæ; the type G of the actual parameter is a subtype of the formal one D and the result-type of a island call $\overline{\mathtt{E}}$ is a subtype (in width) of the result-type of firing all pæninsulæ cape $\overline{\mathtt{E}}, \overline{\mathtt{F}}$: this is sound since some cape will not fire because of pattern-matching failure;
• (Type·Arrow) This rule is used to type a rewrite rule (a.k.a. a pæninsula): the free-variables types of $\mathtt{e}_1$ and $\mathtt{e}_2$ are already "charged" in the context Γ;
• (Type·Set) This rule is a classical type rule to give a cartesian-type to an island composed by all well-typed pæninsulæ;
• (Meth·Ok·Class·Island) Recall that G is the type of the formal pattern. First, using our pluggable type system mechanism, we type-check the island, *i.e.* all pæninsulæ, in the enhanced type system $\vdash_{\mathsf{philo}}$, implemented by $\mathtt{e}_{\mathtt{ile}}$, producing a cartesian-type $\overline{\mathtt{F}}$. Then we call the auxiliary function *arr* on $\overline{\mathtt{F}}$ to get the island's function-type $\mathtt{G} \rightarrow \overline{\mathtt{H}}$. Finally we check that the calculated island result-type $\overline{\mathtt{H}}$ is (point-wise) a subtype of the declared island result-type E.

5.1.6 Motifs et anti-motifs / Patterns and Antipatterns

Dans mon langage jouet basé sur la réécriture Snake [17,59] on a introduit et expérimenté des formes sophistiquées d'algorithmes de filtrage et d'*anti-filtrage* sur des *anti-patterns* : intuitivement un anti-pattern $\backslash\mathtt{e}$ qui se produit dans un cape d'une pæninsula filtrera avec tout paramètre actuel qui ne filtrera pas avec e. Comme on pourrait imaginer, la présence des anti-patterns complique les algorithmes de filtrage, mais augmente l'expressivité et la clarté des pæninsulæ et des îles. Récemment, un sous-ensemble des algorithmes d'anti-filtrage de Snake a été formalisé et implanté dans TOM [KKM07].

In my toy rewriting-based language Snake [17,59] we have introduced and experimented some sophisticated algorithms of antipattern matching: intuitively an antipattern $\backslash\mathtt{e}$ can be provided as a cape of a pæninsula and its behavior is simply that it matches with any actual parameter $\mathtt{e}'$ that does not filter with e. Of course, the introduction of antipatterns complicates the matching algorithm theory but can greatly increase the conciseness of the pæninsulæ. Recently, part of Snake's anti pattern-matching algorithm has been formalized and implemented in TOM [KKM07].

```
class Logic_Table extends Object
{;super()
 bool² and(bool,bool x,bool y){return   (true,true) -> true,
                                       \(true,true) -> false;}
 bool²  or(bool,bool x,bool y){return (false,false) -> true,
                                      \(false,false) -> false;}
 bool         not(bool,bool x){return            \x -> x;}}

t = new Logic_Table();
t.not(and(t.or(true,false),t.and(false,false)))
->not(and(true,false)) ->not(false) ->true          of type bool supertype of bool²
```

Table 5.11: Une classe de Philo Peter avec anti-patterns / Philo Peter's Class with Antipatterns

5.2 L'île impérative / The Imperative Island

Les îles de Philo Peter sont généralement fonctionnelles : en fait elle sont directement héritées par le formalisme des Systèmes de Réécriture ; néanmoins, l'expérience de ML nous a montré que l'ajout d'une forme restreinte et sûre de références dans un paradigme déclaratif s'est révélé très utile dans la pratique. Pour cela, on a ajouté aux îles de Philo Peter le strict minimum pour avoir des références typées dans les îles et grâce à la puissance du filtrage, on peut considérer l'accès en mémoire comme une forme dérivée. L'extension, appelée Ele(phant) Peter, a la syntaxe suivante :

The island are usually functional: in fact they are directly inherited by the formalism of Term Rewriting Systems; nevertheless the ML experience has showed that adding a restricted and sound form of references inside a pure declarative paradigm can be very useful in practice. Having this in mind, we have added a minimum amount of sound and typed reference inside islands and, thanks to pattern-matching, we got for free typed and sound dereferencing. The language extension, called Ele(phant) Peter, has the following syntax:

$\mathtt{e}$	$::=$	$\iota \mid \mathtt{ref\,e} \mid \mathtt{e <- e} \mid \mathtt{!e} \mid \mathtt{e;e}$	Imperative idioms
$\mathtt{!e}$	$\triangleq$	$(\mathtt{ref\,x \text{ -> } x})\,\mathtt{e}$	is a derived idiom
$\mathtt{e_1;e_2}$	$\triangleq$	$(\mathtt{x \text{ -> } e_2})\,\mathtt{e_1} \qquad \mathtt{x} \notin \mathsf{Fv}(\mathtt{e_2})$	is a derived idiom
$(\mathtt{x_0;\ldots;x_n})\,\mathtt{<-}\,(\mathtt{e_0;\ldots;e_n})$	$\triangleq$	$\mathtt{x_0 <- e_0;\ldots;x_n <- x_n}$	is an alias
$\mathtt{let\ e_0 <\# e_1\ in\ e_2}$	$\triangleq$	$(\mathtt{e_0 \text{ -> } e_2})\,\mathtt{e_1}$	is an alias
$\mathtt{neg}$	$\triangleq$	$(\mathtt{true \text{ -> } false}, \mathtt{false \text{ -> } true})$	is an alias
$\mathtt{if\ e\ then\ e_1\ else\ e_2}$	$\triangleq$	$(\mathtt{true \text{ -> } ((x \text{ -> } e_1)\,dummy)},$ $\mathtt{false \text{ -> } ((x \text{ -> } e_2)\,dummy))\,e}$	is an alias

Table 5.12: Syntaxe pour les îles impératives en Ele Peter/ Ele Peter's Syntax for Imperative Islands

Dans une «coquille de noix» :
• *(Emplacement-Termes)*, une emplacement de

In a nutshell:
• *(Location-Terms)* A location-term, denoted

mémoire ι est calculé à partir d'un ensemble dénombrable d'emplacements de mémoire ;
• *(Référence-Termes)* Le terme `ref e` est un pointeur qui référence `e` ; si `e` est un terme de type `C`, alors `ref C` est un pointeur sur `e` de type `C ref` ;
• *(Affectation-Termes)* Le terme $\mathtt{e_1}$ `<-` $\mathtt{e_2}$ est une expression qui assigne la valeur obtenue par l'évaluation de $\mathtt{e_2}$ à l'emplacement obtenu par l'évaluation de $\mathtt{e_1}$.

Comme effet bénéfique de l'ajout des emplacements mémoires dans l'algorithme de filtrage, on obtient que le terme `!e` qui accède à la mémoire est *dérivable par filtrage*. Cela montre la puissance de l'algorithme de filtrage dans un calcul déclaratif à base de règles. Comme d'habitude, la séquence est dérivable en utilisant une simple application à un terme «bidon». Dans le tableau on présente également comment dériver des affectations multiples, des «let» et des conditions en utilisant un encodage classique du produit cartésien et quelques constantes, comme `true`, `false` et `dummy`. Dans la condition, on remarquera que les pæninsulæ `then` et `else` sont «enrobées» dans une abstraction, où `x` n'apparaît pas dans $\mathtt{e_1}$ et $\mathtt{e_2}$, *i.e.*, $\mathtt{x} \notin \mathsf{Fv}(\mathtt{e_1}) \cup \mathsf{Fv}(\mathtt{e_2})$: cela à cause de la stratégie d'appel par valeur de Ele Peter.

by ι, is a store location, taken from a denumerable set of store locations;
• *(Reference-Terms)* The term `ref e` is a referencing term (the-location-of); if `e` is a term of type `C`, then `ref C` is a pointer to `e` of type `C ref`;
• *(Assignment-terms)* The term $\mathtt{e_1}$ `<-` $\mathtt{e_2}$ is an assignment expression, which returns as result the value obtained by evaluating $\mathtt{e_2}$.

As an immediate benefit of the built-in powerful new pattern-matching algorithm, the classical dereferencing term (goto-memory), denoted by `!e`, where `e` is a pointer in the store, can be *easily derived*. This is a quite powerful feature showing the advantages of having built-in pattern matching in a rule-based calculus. As usual, sequencing can be derived with a simple dummy application. Deriving multiple assignments can be useful as well. "Let" and "conditionals"can be easily encoded by using pairing and application (`true`, and `false` and `dummy` are constants). In the conditional, note that the `then` and the `else` pæninsulæ are wrapped in a dummy abstraction, and `x` is fresh in $\mathtt{e_1}$ and $\mathtt{e_2}$, *i.e.*, $\mathtt{x} \notin \mathsf{Fv}(\mathtt{e_1}) \cup \mathsf{Fv}(\mathtt{e_2})$: this is because of the fixed call-by-value strategy of Ele Peter.

5.2.1 Règles clés pour l'exécution / Key Run-time Rules

Ajouter des aspets impératifs à une île implique de fixer une stratégie d'évaluation, normalement un appel par valeur et une évaluation de gauche à droite des expressions. Une fonction globale, appelée mémoire (`S`), des emplacements vers les valeurs ($\mathtt{e_v}$), sera incluse dans la définition de la sémantique opérationnelle. Une mémoire est une fonction partielle de l'ensemble $\mathcal{L}$ des emplacements vers l'ensemble des valeurs $\mathcal{V}al \subset \mathcal{E}xpr$, *i.e.* $\mathtt{S} \in \mathcal{S}tore \simeq [\mathcal{L}oc \Rightarrow \mathcal{V}al]_\perp$. Les emplacements sont des valeurs. On écrira une extension d'une mémoire par $\mathtt{S}[\iota \mapsto \mathtt{e_v}]$ avec la signification suivante :

Adding imperative features to an island means fixing an evaluation strategy, normally a call-by-value, left-to-right evaluation of expressions. An extra partial (non first-class) global function, called *store*, denoted by `S`, from store locations to values (denoted by $\mathtt{e_v}$), must be included in the operational semantics. A store is a partial function from the set $\mathcal{L}$ of locations to the set of values $\mathcal{V}al \subset \mathcal{E}xpr$, c-à-d., $\mathtt{S} \in \mathcal{S}tore \simeq [\mathcal{L}oc \Rightarrow \mathcal{V}al]_\perp$. As usual, location are values. We denote the extension of a store by $\mathtt{S}[\iota \mapsto \mathtt{e_v}]$ with the following meaning:

$$\mathtt{S}[\iota \mapsto \mathtt{e_v}](\iota') \quad \triangleq \quad \begin{cases} \mathtt{e_v} & \text{if } \iota \equiv \iota' \\ \mathtt{S}(\iota') & \text{otherwise} \end{cases}$$

Table 5.13: La fonction mémoire en Ele Peter / Ele Peter's Store Function

La sémantique opérationnelle de Ele Peter reçoit comme argument une expression à évaluer *et* une mémoire $\mathtt{S}$ et produit comme résultat un valeur *et* une nouvelle mémoire $\mathtt{S}'$ modifiée. La relation en une étape ressemble à

$$\mathtt{e}_1 \cdot \mathtt{S} \longrightarrow \mathtt{e}_2 \cdot \mathtt{S}'$$

et ses règles sont :

Ele Peter's operational semantics takes as argument an expression to be evaluated *and* a store $\mathtt{S}$, and produces as output a value *and* a modified store $\mathtt{S}'$. The one-step relation looks like

$$\mathtt{e}_1 \cdot \mathtt{S} \longrightarrow \mathtt{e}_2 \cdot \mathtt{S}'$$

and the extra rules are as follows:

$$\frac{\iota \notin \mathsf{Dom}(\mathtt{S})}{\mathtt{ref}\ \mathtt{e_v} \cdot \mathtt{S} \longrightarrow \iota \cdot \mathtt{S}[\iota \mapsto \mathtt{e_v}]}\ (\text{Run·Ref}) \qquad \frac{\iota \in \mathsf{Dom}(\mathtt{S})}{!\,\iota \cdot \mathtt{S} \longrightarrow \mathtt{S}(\iota) \cdot \mathtt{S}}\ (\text{Run·DeRef})$$

$$\frac{\iota \in \mathsf{Dom}(\mathtt{S})}{\iota\ \mathtt{<-}\ \mathtt{e_v} \cdot \mathtt{S} \longrightarrow \mathtt{e_v} \cdot \mathtt{S}[\iota \mapsto \mathtt{e_v}]}\ (\text{Run·Assign}) \qquad \frac{\mathtt{e}_1 \cdot \mathtt{S} \longrightarrow \mathtt{e}_2 \cdot \mathtt{S}}{\mathtt{C}[\mathtt{e}_1] \cdot \mathtt{S} \longrightarrow \mathtt{C}[\mathtt{e}_2] \cdot \mathtt{S}}\ (\text{Congr·Store·Lift})$$

Table 5.14: Règles clés pour l'exécution des îles impératives dans Ele Peter / Ele Peter's Key Run-time Rules for Imperative Expressions

Dans une «coquille de noix» :
• (Run·Ref) Cette règle écrit dans la mémoire une valeur $\mathtt{e_v}$ dans une emplacement fraîs ι ;
• (Run·DeRef) Cette règle (dérivée) lit dans la mémoire à l'emplacement ι ;
• (Run·Assign) Cette règle affecte les variables : la valeur $\mathtt{e_v}$ et une mémoire modifiée sont rendues comme résultat ;
• (Congr·Store·Lift) Cette règle ajout contextuellement dans toutes les règles la mémoire au valeurs. $\mathtt{C}[\cdot]$ est un contexte avec appel par valeur qui évalue les expressions de gauche à droite.

In a nutshell:
• (Run·Ref) This rule stores a value $\mathtt{e_v}$ into a "fresh" location ι;
• (Run·DeRef) This derivable rule reads the store at the location ι;
• (Run·Assign) This rule performs variable assignment: the value $\mathtt{e_v}$ is stored at location ι: both the value $\mathtt{e_v}$ and a new modified store are produced as output;
• (Congr·Store·Lift) This rule lifts the context rule to value-store pairs. $\mathtt{C}[\cdot]$ is a usual call-by-value applicative context evaluating expressions from left-to-right.

5.2.2 Règles clés pour le typage / Key Type-checking Rules

Typer une île impérative est plutôt simple : le type des pointeurs $\mathtt{C}\ \mathtt{ref}$ est ajouté à la syntaxe des types ; ce type pourra également apparaître dans un contexte de types. Les règles de typage sont :

Typing an imperative island is quite simple: a simple reference-type $\mathtt{C}\ \mathtt{ref}$, the type of a pointer to $\mathtt{C}$, is added to the set of legal types; the context Γ must also contain associations $\iota{:}\mathtt{C}$. The extra rules are:

$$\frac{\iota{:}\mathtt{C} \in \Gamma}{\Gamma \vdash_{\mathsf{ele}} \iota : \mathtt{C}}\ (\text{Type·Loc}) \qquad \frac{\Gamma \vdash_{\mathsf{ele}} \mathtt{e}_1 : \mathtt{C}\ \mathtt{ref} \quad \Gamma \vdash_{\mathsf{ele}} \mathtt{e}_2 : \mathtt{C}}{\Gamma \vdash_{\mathsf{ele}} \mathtt{e}_1\ \mathtt{<-}\ \mathtt{e}_2 : \mathtt{C}}\ (\text{Type·Assign})$$

$$\frac{\Gamma \vdash_{\mathsf{ele}} \mathtt{e} : \mathtt{C}}{\Gamma \vdash_{\mathsf{ele}} \mathtt{ref\ e} : \mathtt{C\ ref}}\ (\text{Type·Ref}) \qquad \frac{\Gamma \vdash_{\mathsf{ele}} \mathtt{e} : \mathtt{C\ ref}}{\Gamma \vdash_{\mathsf{ele}} \mathtt{!\,e} : \mathtt{C}}\ (\text{Type·DeRef})$$

$$\frac{\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D}\ \{\ldots\} \quad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \overline{\mathtt{E}}) \quad \overline{\mathtt{x}}{:}\overline{\mathtt{C}}, \mathtt{this{:}C} \vdash \mathtt{usetype}\ \{\mathtt{e_{imp}}\}\ \mathtt{in}\ \{\mathtt{e_0}\} : \overline{\mathtt{F}} \quad \mathit{arr}(\overline{\mathtt{F}}) = \mathtt{G} \to \overline{\mathtt{H}} \quad \overline{\mathtt{H}} <: \overline{\mathtt{E}}}{\overline{\mathtt{E}}\ \mathtt{m}(\mathtt{G}, \overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e_0;}\}\ \mathtt{OK\ IN\ C}}\ (\text{Meth·Ok·Class·Imp·Island})$$

Table 5.15: Les règles clés de typage de Ele Peter / Ele Peter's Key Typing Rules

Dans une «coquille de noix» :
• (Type·Loc) Cette règle vérifie le type du pointeur ;
• (Type·Assign) Cette règle vérifie l'affectation: si la cible $\mathtt{e_1}$ a le type `C ref`, alors on pourra lui affecter une valeur $\mathtt{e_2}$ de type `C` ;
• (Type·Ref) Cette règle dit que si une expression a le type `C`, alors un pointeur vers cette expression aura le type `ref C` ;
• (Type·DeRef) Cette règle est dérivable : si `e` est un pointeur sur une expression de type `C`, alors son accès en mémoire `!e` est de type `C` ;
• (Meth·Ok·Class·Imp·Island) Cette règle ressemble à la règle (Meth·Ok·Class·Imp·Island) mais le corps de l'île est typé dans le système de types $\vdash_{\mathsf{ele}}$, implanté par l'expression $\mathtt{e_{imp}}$. Grâce au système de types pluggables de Ele Peter, les références ne seront permises qu'à l'intérieur des îles et non en dehors.

In a nutshell:
• (Type·Loc) This rule just checks whether the location has a fixed type;
• (Type·Assign) This rule deals with variable assignment: the only possible choice is to assign an object $\mathtt{e_2}$ of type `C` to an expression $\mathtt{e_1}$, of type `C ref` ;
• (Type·Ref) This rule says that, if an object `e` has type `C`, then a pointer to this object, denoted by `ref e`, has type `C ref`.
• (Type·DeRef) This rule is derivable and it says that, if `e` is a pointer to an object of type `C`, then its access in memory, denoted by `!e`, has type `E`;
• (Meth·Ok·Class·Imp·Island) This rule behaves as the original rule (Meth·Ok·Class·Island) but the body of the island is typed in a type system $\vdash_{\mathsf{ele}}$, implemented by $\mathtt{e_{imp}}$. Thanks to Ele Peter's "type-pluggability", references are allowed only inside islands.

5.2.3 Un exemple plus compliqué / A More Tricky Example

Cette île impérative calcule une forme normale négative. On présente deux codages différents : dans le premier codage, la fonction est partagée au travers d'un pointeur et la récursivité est obtenue à travers un accès dans la mémoire. Dans le second codage, les formules logiques sont partagées avec des pointeurs vers les sous-arbres. Le nouveau langage logique contient des pointeurs vers les formules. Dans les deux encodages, la variable `self` joue le rôle de la méta-variable `this` de Ele Peter. Cet exemple pourrait être implanté en utilisant le `this` standard de Baby Peter pour capturer la récursivité à travers les fonctionnalités basiques de Baby Peter (mais au détriment de l'efficacité en temps et en occupation mémoire). Finalement, les deux classes avec les îles impératives

This imperative island computes a negation normal form. We present two imperative encodings: in the first one, the function is shared via a pointer and recursion is achieved via dereferencing. In the second one, formulas are shared also with back-pointers to shared sub-trees. The new processed input is an implication-free language with pointers to formulas. In both encoding, the variable `self` plays the role of Baby Peter's metavariable `this`. Of course, this example can be implemented using `this` to capture recursion via the basic Baby Peter's functionalities (but this would strongly affect performance in time and space). Finally, the two classes with the imperative islands are defined as follows, where `formulas` is the type of formulas:

peuvent être définies (on suppose que `formulas` est le type basique des formules) :

```
bool  ::= true | false    flag ::= boolean ref    palg ::= ref alg
alg   ::= alg | and (flag,palg ,palg ) | or(flag,palg ,palg ) | not(flag,palg )

class Imperative_Logic_Island_1 extends Object
{;super();
 bool[6] nnf(bool,formulas x){return let self < ref dummy in
                                   let tmp  < impisland_1 in self<-tmp;(nnf x);}}


class Imperative_Logic_Island_2 extends Object
{;super();
 bool[6] nnf(bool,formulas ref x){return
                                   let self < ref dummy in
                                   let tmp  < impisland_2 in self<-tmp;(nnf x);}}
```

Table 5.16: Deux classes avec des îles impératives / Two Classes with Imperative Islands

où $\texttt{impisland}_1$ est : | where $\texttt{impisland}_1$ is:

```
                    alg_1  ->  alg_1
                    alg_2  ->  alg_2
               not(not(x))  ->  !(self x),
              not(or(x,y))  ->  and (!(self not(x))),!(self not(y)),
            not(and (x,y))  ->   or(!(self not(x))),!(self (not(y)),
               and (x,y))   ->  and (!(self x),!(self y)),
                  or(x,y)   ->   or(!(self x),!(self y))
```

Table 5.17: Partage de la fonction mais pas des données / Sharing the Function but not the Data

et $\texttt{impisland}_2$ est : | and $\texttt{impisland}_2$ is:

```
                        ref alg_1  ->  ref alg_1
                        ref alg_2  ->  ref alg_2
          not(b_1,ref (not(b_2,x)))  ->  !(self (!x)),
         not(b_1,ref (or(b_2,x,y)))  ->  and (ref false,
                                         !(self (ref (not(ref false,x)))),
                                         !(self (ref (not(ref false,y))))),
```

```
not(b1 , ref (and (b2 , x , y)))   ->   or(ref false ,
                                        ! (self (ref (not(ref false , x)))) ,
                                        ! (self (ref (not(ref false , y))))) ,
              and (b , x , y)      ->   if (neg ref bool)
                                        then (b , x , y) <- (true , ! (self (! x)) , ! (self (! y)))
                                        else (and (b , x , y)) ,
               or(b , x , y)       ->   if (neg ref bool)
                                        then (b , x , y) <- (true , ! (self (! x)) , ! (self (! y)))
                                        else (or (b , x , y))
```

Table 5.18: Partage de la fonction et des données / Sharing Function and Data

Sources d'inspiration / Inspiration Sources

Ce chapitre a été inspiré par tous les articles que j'ai écrits avec Claude, Horatiu et Benjamin à Nancy et avec Bernard à Sophia depuis 2000 (quasiment toute l'histoire dans notre livre sur le Calcul de Réécriture [65], voir ma page web pour la liste des chapitres). Mes articles les plus pertinents sont [4,17,20,21,22,24,27, 28,29,30,50,53,59,60] : les plus pédagogiques sont [4,27,30].

This chapter has been influenced by all the work I did with Claude, Horatiu, and Benjamin during my stay in Nancy and with Bernard during my stay in Sophia since 2000 (almost all the story in the forthcoming book on the Rewriting Calculus [65], see on my home page for the index). My most pertinent papers are [4,17,20,21,22,24,27,28,30,50,53,59,60]: the most pedagogical are [4,27,30].

Chapter 6

Peter devient sage / Peter Becomes Wiser

Ce chapitre est dédié à Furio : grâce au `call/cc`*, aux sémantiques à la Scott, Kahn et Plotkin tu m'as fait comprendre que l'informatique théorique est un art et qu'il pouvait être mon métier.* Arrivederci a Roma...

This chapter is dedicated to Furio: thanks to `call/cc`*, and Scott/Kahn/Plotkin's Semantics you helped me to realize that Theoretical Computer Science is almost an art, and that it could be my job as well.* Arrivederci a Roma...

6.1 Peter mélange algorithmes et preuves / Peter Mixes Algorithms with Proofs

Ce chapitre nous parle de l'âge de la sagesse de Peter... La dernière extension qu'on présente dans cette thèse d'habilitation, appelée Sage Peter, mélange algorithmes et preuves de correction d'algorithmes. Ceci devrait être *le* paradigme de programmation dans un futur proche : on pense que certains choix de conception adoptés dans Peter pourraient être utiles pour les prochains langages de programmation réels et leurs extensions.

This chapter will narate about the Peter's age of sageness... The last extension we present in this Habilitation Thesis, called Wise Peter, deals with the possibility of mixing algorithms and correctness proofs of such algorithms. This should be *the* programming paradigm in a near future: we believe that some design choice adopted in Peter will be useful for the next real-size programming language and their sharp extensions.

6.1.1 Syntaxe / Syntax

On étend la syntaxe des méthodes et des classes avec des constructions auxiliaires qui permettent d'inclure un *certificat* de correction de l'algorithme : le certificat pourra être vérifié à tout moment dans un «cadre logique» (le *Logical Framework*, LF) fondé sur l'isomorphisme de Curry-Howard.

We extend Peter's class and method syntax with auxiliary constructions that allow us to include a *proof certificate* (witnessing the soundness of the code), which can be verified in a Logical Framework LF based on Curry-Howard's isomorphism.

CL	::=	class C extends D{$\overline{\text{C}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$} [with cert C]	Class Declarations
$\mathcal{CL}$	::=	cert C {C; [e]} | cert C extends D {$\overline{\text{C}}$ $\overline{\text{m}}$}	Certificate Declarations
M	::=	C m($\overline{\text{C}}$ $\overline{\text{x}}$){return e;} [with cert C]	Methods
σ	::=	a | Πx:σ.σ | λx:σ.σ | σ e	LF−types (Propositions)
K	::=	Type | Πx:σ.K | λx:σ.K | K e	LF−kinds (Constructors)
e	::=	alg | x | λx:σ.e | e e	LF−terms (Proofs)

Table 6.1: La syntaxe de Sage Peter / Wise Peter's Syntax

Dans une «coquille de noix» :
• CL est une fonction partielle des noms de classes vers des déclarations de classes : notons que la déclaration d'une classe peut être enrichie avec son certificat de correction, c-à-d. un «certificat de garantie» que le code dans la classe respecte les spécifications;
• $\mathcal{CL}$ est une fonction partielle des noms de certificats vers les déclarations de certificats. Un certificat C peut être un simple théorème σ dans LF avec son terme de preuve e dans LF : si e est absent, c-à-d. cert C {σ; } alors le certificat est un *axiome*, c-à-d. un théorème qui n'a pas encore été prouvé, dont la preuve est reportée à la phase de compilation à travers la résolution d'un problème d'«habitation de type» dans LF. Un certificat peut également être une extension d'un autre certificat (parent) avec des certificats de méthodes supplémentaires ;
• M désigne une définition de méthode qui est maintenant suivie d'un certificat optionnel C qui garantissant que le code respecte sa spécification formelle ;
• σ sont des *types* en LF (*familles* dans le jargon de LF) ; ils représentent, dans l'isomorphisme de Curry-Howard, des théorèmes à prouver ;
• K sont des *constructeurs* en LF, utilisés pour construire des types dépendants ;
• e sont des termes en LF (*objets* dans le jargon de LF) ; ils représentent, dans l'isomorphisme de Curry-Howard, des preuves de théorèmes, c-à-d. que e : σ signifie informellement que «*le théorème σ est prouvé par* e».

In a nutshell:
• CL is a partial function from class names to class declarations: note that a class declaration now comes with an optional *certificate*, that is a guarantee that all code inside the class meets the specifications;
• $\mathcal{CL}$ is a partial function from certificate names to certificate declarations. A certificate C can be a simple LF-theorem σ equipped with some LF proof-term e: if e is absent (*i.e.*, cert C {σ; }), then the certificate is an *axiom*, *i.e.* a not yet proved theorem whose proof will be postponed in the compilation phase solving an inhabitation problem in LF. A certificate can be also an extension of another (super) certificate with additional or overridden *method certificates*;
• M are method definitions that now come with an optional certificate C that guarantees that code meets a formal specification;
• σ are LF types (*families* in LF's jargon); they represent, in the Curry-Howard isomorphism, theorems to inhabit;
• K are LF *constructors* used to build dependent-types.
• e are LF term (*objects* in LF's jargon); they represent, in the Curry-Howard isomorphism, proof for theorems, *i.e.* e : σ means informally *"theorem σ is proved by* e*"*.

6.1.2 Intermezzo / Intermezzo

On peut se poser la question :
Quelle est la signification de

One question arise:
What is the meaning of

l'héritage dans la théorie de la preuve?
L'intuition la plus simple d'un certificat

`cert C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{m}}\}$

est que le certificat `C` est une extension du certificat `D` avec des preuves de nouvelles méthodes ou des nouvelles preuves de méthodes qui sont déjà présentes dans `D`. À travers le principe, orienté objet, de substitution, si `e` est une preuve du théorème `C` et si le théorème `C` est un sous-type du théorème `D` (noté par `C <: D`), alors une preuve `e` de `C` pourrait être utilisée chaque fois qu'on attend une preuve de `D`. Donc, le sous-typage améliore la *réutilisation des preuves*.
Comme exemple de réutilisation de preuves, on considère le certificat de Sage Peter :

`cert C extends D { }`

qui reproduit la technique mathématique bien connue qui consiste à prouver un *théorème plus fort*, c-à-d. une preuve d'un autre théorème qui implique naturellement la preuve du théorème originel ; la réutilisation des preuves a quelque analogie avec le sous-typage en *largeur* dans les langages objets.
Comme autre exemple de réutilisation de preuves, on considère le certificat de Sage Peter

`cert C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{m}}\}$

qui dit que si on peut prouver un certificat `D` contenant des théorèmes et si on peut prouver d'autres théorèmes étiquetés, soit $\overline{\texttt{C}}\ \overline{\texttt{m}}$, alors on peut prouver un plus gros *sous-certificat* `C` ; en utilisant une relation orientée objets qui ressemblerait au *sous-typage en largeur* ou une règle d'élimination de *types-intersection* dans la logique propositionnelle, le certificat `C` pourrait être utilisé dans tout contexte qui attend un certificat `D` plus petit.

inheritance in proof theory?
The most simple intuition of certificate

`cert C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{m}}\}$

is that the certificate `C` is an extension of the certificate `D` with new fresh method-proofs or new proofs for methods which are already present in `D`. Using the classical object-oriented substitution principle, if `e` is a proof for the theorem `C`, and the theorem `C` is a subtype of the theorem `D` (denoted by `C <: D`), then a proof `e` of `C` can be used every time a proof of `D` is required. Thus, subtyping improve *proof-reuse*.

As a first simple example of proof reuse, consider the Wise Peter's certificate

`cert C extends D { }`

that capture the well-known technique in mathematics of proving a *stronger statement*, *i.e.* a proof of another theorem that naturally implies the original theorem; proof reuse is somewhat reminiscent to *depth* subtyping in object-oriented languages.

As an another simple example of proof reuse, consider the Wise Peter's certificate

`cert C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{m}}\}$

saying that if we can prove a certificate `D` containing some theorems, and if we can prove other labeled theorems, say $\overline{\texttt{C}}\ \overline{\texttt{m}}$, then we can prove a bigger *sub-certificate* `C`; using a *width-subtyping*-object-oriented relation, or an *intersection-type*-like elimination rule in propositional logic, the certificate `C` can be used in any context expecting a smaller certificate `D`.

6.1.3 Programmer avec des preuves : pourquoi est-ce si complexe ? / Programming with proofs: why so complex?

La difficulté de *programmer et prouver*, en utilisant l'isomorphisme de Curry-Howard, est que le langage algorithmique (Peter, un simple langage objet avec typage et sous-typage nominal) et le langage des preuves (LF, le cadre logique, c-à-d. un λ-calcul typé avec des types dépendants) sont très différents et déconnectés l'un de l'autre ; pour prouver des propriétés de correction d'un code source, plusieurs directions sont possibles et applicables dans la version actuelle et dans les futures versions de Peter :

The difficulty in *programming with proofs*, using Curry-Howard isomorphism, is that the algorithmic source language (Peter, a simply typed object-oriented language with nominal typing and subtyping) and the proof-language (LF, the Logical Framework, *i.e.* a typed lambda calculus with dependent-types) are quite different and disconnected from each other; in order to prove some correctness properties on the source code, several directions are plausible and applicable in the actual and in future Peter's versions:

• (Peter V1 + LF). Ceci est la méthodologie la plus «économique» pour certifier des programmes ; le programmeur écrit un corps e de méthode et la proposition σ qui concerne la propriété qu'on voudrait prouver ; après il présente une traduction *adéquate* du corps e dans LF, avec des *obligations des preuves* obtenues en analysant le code e ; finalement il construit un certificat pour une classe en cherchant un «habitant» e_0 de σ pour chaque méthode définie dans une classe. Si tous les habitants sont trouvés, alors un certificat pour la classe pourra être montré. Une *adéquation* entre Peter- et LF- codes sera montrée sur papier.

• (Peter V1 + LF + Peter's **Assertion Compiler).** Cette direction va un peu plus loin que la précédente. Le programmeur décore les corps des méthodes avec des *annotations*, écrites dans un langage logique du premier ordre ou, dans le cas de Ele Peter, en utilisant des pre-post conditions *à la* Floyd-Hoare. Ces annotations sont manipulées par un *Compilateur d'Annotations*, dans le but de produire un ensemble d'*obligations de preuves* qui seront prouvées dans l'assistant à la preuve qui implante le cadre logique ; le compilateur d'annotations de Peter traduira également le code orienté-objets dans un langage intermédiaire qui pourrait être interprété par un assistant de preuve. Ensuite, la recherche (automatique ou semi-automatique) peut commencer et, pendant cette phase, le programmeur peut découvrir des erreurs dans l'algorithme ou dans les annotations et modifier le code orienté-objets ou les annotations. Il s'agit d'un processus qui peut être réitéré plusieurs fois (cela ressemble à un raffinement dans B [Abr96] ou à un calcul de point fixe). Lorsque la recherche se termine, un *témoin* de preuve (ou *habitant*) de la proposition pourra être montré et la classe sera certifiée. Observons que la syntaxe actuelle de Peter n'oblige pas une classe à d'être certifiée *in toto*: il est donc possible de ne certifier qu'une méthode à l'intérieur d'une classe non certifiée.

• (Peter V1 + Peter**-compiler).** Cette direction, que j'appellerai *programmer avec des objets et prouver*, essaye d'introduire une méthodologie orientée objets dans un cadre logique fonctionnel pur. Notre objectif est d'avoir un *langage objets universel* pour décrire preuves et algorithmes. Sage Peter possède des expressions inspirées par LF ainsi que par Java.

• (Peter V1 + LF). This is the most "economic" direction to certify programs; the programmer writes the method body e and the proposition σ concerning the property we want to prove; then he provides an *adequate* translation of the body e into LF, together with *proof obligations* obtained by analyzing the code e; finally he builds the class-certificate by finding an inhabitant e_0 for σ for each method defined in the class. If all inhabitants are found, then a class-certificate can be showed. Adequacy between Peter- and LF-codes is proved with pencil and paper.

• (Peter V1 + LF + Peter's **Assertion Compiler).** This direction goes a step further than the previous one. The programmer first decorates method bodies with *annotations*, written in a first-order assertion language or, in the case of Ele Peter, using pre-post conditions *à la* Floyd-Hoare. Those annotations are then processed by an *Assertion Compiler* in order to produce a *set of obligations* that will be proved in the proof-assistant implementing the Logical Framework; the Peter Assertion Compiler also translates the object-oriented code into an intermediate meta-language that can be understood by the proof-assistant. Then, the (automatic or semi-automatic) proof development starts, and during this phase, the programmer can discover mistakes in the algorithm or in the assertions, and modify either the object-oriented code or the code annotations, accordingly. This process can iterate "back-and-forth" (this is reminiscent of a B-refinement [Abr96], or of a fixpoint computation). Once the development is complete, a *witness* (an inhabitant) of the proposition can be showed, and the class is certified. Note that the the actual Peter syntax does not enforce the certification *in toto* of a class: in fact, it is possible to certify just one method inside a non certified class.

• (Peter V1 + Peter**-compiler).** This direction, that I will call *object-oriented programming with proofs*, tries to push some object-oriented methodologies inside a pure functional Logical Framework. Our vision is to have an *universal object-oriented language* to describe algorithms and proofs. Wise Peter has LF-inspired constructors as well as Java-inspired

Un compilateur pour Sage Peter devra inclure une phase de traduction interne du code orienté objets annoté dans un métalangage. La phase de compilation devra inclure la phase de développement de la preuve et, dans le cas d'une habitation de la proposition, le code sera certifié. On pourrait même envisager une *extraction* du contenu algorithmique de la preuve. Finalement, la possibilité d'avoir un *compilateur certifié* pour le langage source ou le code extrait devrait «clôturer» la *chaîne de certification* en produisant du code binaire certifié qui pourra être fièrement exécuté sur du matériel non certifié... Les extensions futures de Peter (Peter V2) essayeront d'intégrer au mieux des aspects orientés objets dans le cadre logique de LF. Il nous semble possible, dans un futur prochain, d'envisager la tâche de programmer, composée de deux activités principales : réaliser des algorithmes et produire des preuves pour prouver la correction des algorithmes dans un *langage de preuve/programmation universel* unique.

• **(LF + LF-compiler).** La dernière direction, que j'appellerai *programmer dans des preuves*, est orthogonale *à la* précédente et elle ne fait pas partie de la démarche de Peter; néanmoins, elle doit être mentionnée car elle est intéressante et couramment étudiée dans des équipes en France, comme LogiCal, Proval, Gallium et FoCal. Elle se base sur le fait que le langage fonctionnel *à la* ML appartient à la même catégorie des métalangages fonctionnels *à la* LF. Ceci est un argument fort pour écrire des algorithmes dans le paradigme fonctionnel plutôt que dans le paradigme des objets. Dans une «coquille de noix», cette veine de recherche cherche à combler le fossé entre les langages algorithmiques fonctionnels, les métalangages pour la preuve et leurs interprètes et compilateurs. En d'autres termes et contrairement à Peter, le *langage universel* pourrait être fonctionnel. Le programmeur pourrait écrire directement l'algorithme dans un style logico-fonctionnel, avec des annotations, obligations de preuves et propositions à prouver. Actuellement, les assistants de preuve fondés sur l'isomorphisme de Curry-Howard, sont capables d'*extraire* du code fonctionnel en Haskell, ML, Caml et Scheme, à partir de leurs preuves ; notez qu'il existe des logiciels qui traduisent fidèlement du code Caml vers du code Java [CH05, Ler06].

constructors. Wise Peter's compiler will include an automatic internal translation of the object-oriented annotated code into a suitable meta-language. The compilation phase will include the proof development and, in case of successful inhabitation of the proposition, the code will be certified. One could even envisage a *extraction* of the computational content from the proof. Finally, the possibility of having a *certified compiler* for the source or the extracted code will "conclude" the *certification chain*, producing a certified binary code that will be proudly executed on a suitable (uncertified) hardware... Future Peter's extensions (Peter V2) will try to better integrate object-oriented features into the LF logical framework. It is feasible that, in a near future, the task of programming will be composed by two fundamental and strictly coupled activities: design algorithms and design proofs for algorithms into an unique *universal proof/programming language*.

• **(LF + LF-compiler).** This last direction, that I will call *programming inside proofs*, is orthogonal to the previous one and it is not part of Peter's philosophy; nevertheless, it is worth to be mentioned since it is interesting, and currently under deep study by some French project-teams like LogiCal, Proval, Gallium, and FoCal. The direction relies on the fact that functional languages *à la* ML belong to the same category of functional metalanguage *à la* LF. This is a strong argument to write algorithms inside the functional instead of the object-oriented paradigm. In a nutshell, this very active research strand tries to fill the gap between algorithmic languages (possibly functional), proof meta-languages, and their interpreters and compilers. In other words, and in contrast with Peter philosophy, the *universal language* may be a functional one. With this in mind, the programmer can write directly the algorithm in a functional-logic language, together with annotations, proof obligations, and proposition to be proved. A successful interactive compilation would allow to *extract* the computational content from the proof. Actually, Curry-Howard base proof assistants, are able to extract functional code in Haskell, ML, Caml, and Scheme from their proofs; recent experiments have showed also extraction of pure functional Java-code. It is worth noticing that there exists tools that translate faithfully Caml-

code into pure Java [CH05, Ler06].

6.1.4 Intermezzo / Intermezzo

Concernant mes petites expériences de production de logiciels sûrs à travers une certification formelle, je dois beaucoup aux expériences conduites dans le cadre de mon projet INRIA Miró (2001-2003) et, en particulier, dans l'étude d'un langage expérimental et de son interprète, appelé FunTalk [55,56] et un autre appelé iRho [4,20]. FunTalk était un langage objets pur, basé sur le calcul des objets primitifs de Martin et Luca [AC96] ; certaines caractéristiques de FunTalk peuvent être retrouvées dans Fly Peter et dans Sharp Peter ; avec Alberto et Marino nous avons conduit des expériences de certification d'un interprète de la version impérative du calcul des objets en utilisant la *syntaxe d'ordre supérieure* [7,23,25]. iRho est probablement le premier langage basée sur la réécriture avec des aspects impératifs avec mémoire et algèbre de pointeurs. Avec Bernard, nous avons conduit un expérience très intéressante en utilisant une méthodologie qui a inspiré celle de (Peter V1 + LF). Je voudrais vous en raconter un peu plus.

Concerning the experimentation of producing safe software via a formal certification, I am indebted to some experiments I have done in the setting of my INRIA Miró Team (2000-2003), in particular in the study of an experimental language and its interpreter, called FunTalk [55,56] and another called iRho [4,20]. FunTalk was an object-based toy language, based on the Primitive Object Calculus of Martin and Luca [AC96]; some FunTalk's features can be found in Fly Peter and Sharp Peter; with Alberto and Marino we conducted some experiments of certifying an interpreter of a imperative version of the object calculus using *Higher-Order Abstract Syntax* [7,23,25]. iRho was probably the first typed rewriting-based language with some imperative features (*i.e.* with store and pointers algebras). With Bernard, we conducted an interesting experiment of building a safe interpreter using a methodology that inspired the one of (Peter V1 + LF). I want to tell you a little bit about this experience.

À la base de notre recherche, nous sommes d'abord partis sur un «design» de iRho clair et élégant du point de vue mathématique ; nous avons ensuite continué avec une implantation d'un prototype qui respectait les spécifications, pour finalement terminer avec une certification mécanique des propriétés mathématiques du design, en cherchant la plus simple propriété d'adéquation du logiciel par rapport à ses spécifications. Ces trois phases ont été très liées et très souvent chaque choix dans une phase a induit un choix correspondant dans une autre phase, ce qui nous a forcés a revenir en arrière très souvent. Le raffinement de ce processus a été fait d'une façon itérative jusqu'à ce que toutes les propriétés globales recherchées soient obtenues. Chaque phase a influencé l'autre. Comme exemple de la méthodologie utilisée dans la définition d'un interprète pour le calcul de réécriture, on montre, ci-dessus, la définition mathématique pure d'un pas de l'interprète, en sémantique naturelle *à la* Kahn, puis son implantation correspondante en Scheme, puis un codage adéquat dans le cadre logique sous-jacent Coq et, finalement, une proposition logique qui établit que la sémantique opérationnelle est déterministe ; le lecteur pourra apprécier comment les trois

Essentially, we started from a clean and elegant mathematical design of iRho, then we continued with an implementation of a prototype satisfying the design, and finally we completed with a mechanical certification of the mathematical properties of the design, by looking for the simplest adequacy property of the related software implementation. These three phases were strictly coupled and, very often, one particular choice in one phase induced a corresponding choice in another phase, very often forcing backtracking. Refinement of this process was done by iterating cycles until all the global properties wanted were reached. Each phase influenced the other. As an example of the methodology in defining a simple interpreter for the Rewriting Calculus, we show below the pure mathematical definition of a rule application in Kahn's Natural Semantics, then its implementation in Scheme, then its adequate representation in the Logical Framework underpinning Coq, and finally the proposition stating that the natural semantics is deterministic; the reader can see how much the three specifications are close to each other.

spécifications sont très proches l'une de l'autre. Le chemin pour trouver d'une façon automatique des représentations adéquates dans un cadre logique d'algorithmes fonctionnels, impératifs ou à objets n'est pas encore parfait : il reste du travail à effectuer. Ci-dessus, on présente trois facettes d'un fragment du même algorithme (l'interprète de iRho décrit en sémantique naturelle), écrit en Mathématique (et affiché en LaTeX), en Coq et en Scheme.

Much more work would be necessary, by our community, to provide adequate representations or "automatic" translations in the Logical Framework of algorithms described in object-oriented or imperative languages. Below, we presents the three facets of a fragment of the same algorithm (a Kahn's natural semantic interpreter of iRho), written in Mathematics (and displayed in LaTeX), in Coq, and in Scheme.

LaTeX code

$$\frac{\rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \quad \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \quad \vdash \langle A_{\mathsf{v}} \centerdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}}}{\rho \vdash A\, B \Downarrow_{\mathsf{val}} C_{\mathsf{v}}}\,(\mathrm{Red{\cdot}Appl_v})$$

$$\frac{\rho \vdash \langle P \centerdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho' \quad \rho' \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}}}{\vdash \langle \langle P \to A \centerdot \rho \rangle \centerdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}}}\,(\mathrm{Call{\cdot}FunOk})$$

Scheme code

```
(define-generic (Eval::Value expr::Expr env))
(define-method (Eval::Value expr::App env)
 (let ((fun (Eval (App-fun expr) env))
       (arg (Eval (App-arg expr) env)))
 (Call fun arg)))
...
(define-generic (Call::Value fun::Value arg::Value))
... omitted...
```

Coq internal representation

```
Mutual Inductive eval : expr -> env -> value -> Prop :=
 evalApp: (F:expr)(e:env)(f:value)
(eval F e f) -> (A:expr)(a:value)
(eval A e a) -> (v:value)
(call f a v) ->
(eval (App F A) e v)
| ...
with call : value -> value -> value -> Prop := ... omitted...
```

Coq proposition to be inhabited.

```
Theorem eval_deterministic: (A:expr)(e:env)(s:store)(v1:value)
 (eval A e s v1 s1) -> (v2:value) (eval A e s v2 s2) -> (v1=v2).
```

Table 6.2: Trois facettes de la même spécification / Three Facets of the Same Specification

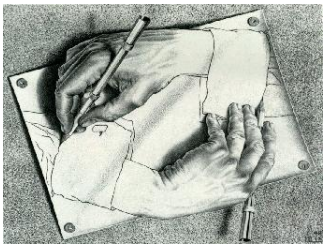


Table 6.3: Un joli cercle vertueux *à la* Escher / A Nice Virtuous Circle *à la* Escher

6.1.5 Règles clés pour l'exécution / Key Run-time Rules

Dans le nouveau paradigme de programmer et prouver, les preuves n'ont aucun contenu calculatoire ; elle doivent juste exister ; donc la sémantique opérationnelle de Sage Peter ne prendra pas en considération la partie preuve (c-à-d. tous les termes en LF) et il exécutera seulement le code orienté-objets pur de Peter. Il n'y a donc pas de nouvelles règles de sémantique dynamique à montrer. On se contentera, pour ceux qui ne sont pas acolytes de LF, de présenter ses règles classiques de réduction :

In the new programming-with-proofs paradigm, proofs need only to be exhibited but not runned; as such, the operational semantics of Wise Peter just discards the proof part (*i.e.* all LF-terms) and only executes pure Peter object-oriented code. Thus, no important rules need to be presented in this section. As a useful reminder for non LF-acolytes, we recall the classical LF-reduction rules:

$$(\beta-\text{Objects}) \qquad (\lambda \mathtt{x}{:}\sigma.\mathtt{e}_1)\,\mathtt{e}_2 \;\mapsto_\beta\; \mathtt{e}_1[\mathtt{e}_2/\mathtt{x}]$$

$$(\beta-\text{Families}) \qquad (\lambda \mathtt{x}{:}\sigma.\sigma)\,\mathtt{e} \;\mapsto_\beta\; \sigma[\mathtt{e}/\mathtt{x}]$$

$$(\beta-\text{Kinds}) \qquad (\lambda \mathtt{x}{:}\sigma.\mathtt{K})\,\mathtt{e} \;\mapsto_\beta\; \mathtt{K}[\mathtt{e}/\mathtt{x}]$$

Table 6.4: Les règles de réduction de LF / LF's Reduction Rules

6.1.6 Règles clés pour le typage / Key Type-checking Rules

Les règles de typage les plus intéressantes de Sage Peter sont décrites ici :

The more interesting type-checking rules of Wise Peter are described below:

$$\frac{\begin{array}{l}\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D}\ \ldots \quad \mathcal{CL}(\mathtt{B}) = \mathtt{cert\ B}\ \{\sigma;\} \\ \overline{\mathtt{x}}{:}\overline{\mathtt{C}}, \mathtt{this}{:}\mathtt{C} \vdash_{\mathsf{baby}} \mathtt{e} : \mathtt{F} \quad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{E}) \quad \mathtt{F} <: \mathtt{E} \\ \exists\ \mathtt{e}_0(\mathtt{C},\mathtt{m}).\ [\ \vdash_{\mathsf{baby}} \mathtt{usetype}\ \{\mathtt{e}_{\mathsf{LF}}\}\ \mathtt{in}\ \{\mathtt{e}_0(\mathtt{C},\mathtt{m})\} : \sigma\ ]\end{array}}{\mathtt{E\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e};\}\ \mathtt{OK\ IN\ C\ with\ cert\ B}}\ (\text{Meth·Ok·Class·Cert}_1)$$

$$\frac{\begin{array}{l}\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D}\ \ldots \quad \mathcal{CL}(\mathtt{B}) = \mathtt{cert\ B}\ \{\sigma; \mathtt{e}_0\} \\ \overline{\mathtt{x}}{:}\overline{\mathtt{C}}, \mathtt{this}{:}\mathtt{C} \vdash_{\mathsf{baby}} \mathtt{e} : \mathtt{F} \quad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{E}) \quad \mathtt{F} <: \mathtt{E} \\ \vdash_{\mathsf{baby}} \mathtt{usetype}\ \{\mathtt{e}_{\mathsf{LF}}\}\ \mathtt{in}\ \{\mathtt{e}_0\} : \sigma\end{array}}{\mathtt{E\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e};\}\ \mathtt{OK\ IN\ C\ with\ cert\ B}}\ (\text{Meth·Ok·Class·Cert}_2)$$

$$\frac{\begin{array}{l}\mathcal{CL}(\mathtt{C}) = \mathtt{cert\ C\ extends\ D}\ \{\overline{\mathtt{B}}\ \overline{\mathtt{m}}\} \\ \mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}}); \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \quad \mathit{meth}(\overline{\mathtt{M}}) = \overline{\mathtt{m}} \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C\ with\ cert}\ \overline{\mathtt{B}}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{with\ cert\ C\ OK}}\ (\text{Class·Ok·Cert})$$

Table 6.5: Les règles de typage les plus intéressantes de Sage Peter / Wise Peter's Most Interesting Typing Rules

Dans une «coquille de noix»:
• (Meth·Ok·Class·Cert$_1$) Cette règle se comporte comme la règle originale de Baby Peter (Meth·Ok·Class) excepté que le certificat `cert B` $\{\sigma;\}$ doit être prouvé. Grâce à la nature constructive du cadre logique LF, l'existence d'un terme LF $\mathtt{e}_0(\mathtt{C},\mathtt{m})$ (qui dépend de la classe `C` et de la méthode `m`) doit être trouvé à travers un problème «d'habitation» du théorème σ ; formellement, le jugement à prouver est $\vdash_{\mathsf{baby}}$ `usetype` $\{\mathtt{e}_{\mathsf{LF}}\}$ `in` $\{\mathtt{e}_0(\mathtt{C},\mathtt{m})\} : \sigma$, où $\mathtt{e}_{\mathsf{LF}}$ est le système de types de LF, écrit en Baby Peter.
• (Meth·Ok·Class·Cert$_2$) Cette règle est différente de (Meth·Ok·Class·Cert$_1$) car le terme LF $\mathtt{e}_0$ qui prouve le théorème LF σ est déjà dans la définition du certificat ; puisque le problème du typage en LF est décidable, la vérification du certificat peut être fait automatiquement pendant la compilation ;
• (Class·Ok·Cert) Cette règle se comporte comme la règle originale de Baby Peter (Class·Ok) à la seule différence que les méthodes sont typées *vis-à-vis* des certificats de sûreté correspondants. Il est aussi intéressant d'observer que les certificats sont nommés comme les classes correspondantes sans que cela crée aucune ambiguïté car ils seront mémorisés dans différentes bases de données (`CL` et $\mathcal{CL}$, respectivement).

In a nutshell:
• (Meth·Ok·Class·Cert$_1$) This rule behaves as an ordinary (Meth·Ok·Class) rule with the exception that the certificate `cert B` $\{\sigma;\}$ needs to be inhabited. Thanks to the constructive nature of the Logical Framework, the existence of an LF-term $\mathtt{e}_0(\mathtt{C},\mathtt{m})$ (depending on class `C` and of method `m`) must be exhibited via an LF-inhabitation of theorem σ; formally, the judgment to prove is $\vdash_{\mathsf{baby}}$ `usetype` $\{\mathtt{e}_{\mathsf{LF}}\}$ `in` $\{\mathtt{e}_0(\mathtt{C},\mathtt{m})\} : \sigma$, where $\mathtt{e}_{\mathsf{LF}}$ is the LF's *type checker*, written in Baby Peter.
• (Meth·Ok·Class·Cert$_2$) This rule differs from the (Meth·Ok·Class·Cert$_1$) rule because the LF-term $\mathtt{e}_0$, proving the LF-type theorem σ, is already provided in the certificate; since type checking in LF is decidable, checking the certificate can be done automatically during type-compilation.
• (Class·Ok·Cert) This rule performs all checks of the original Baby Peter's (Class·Ok) rule with the only difference that methods are *pointwise* checked with their safety certificate. It is also worth noticing that class certificates have the same name as classes itself; this will not cause any ambiguity problem since it will be fetched into in different databases (`CL` and $\mathcal{CL}$, respectively).

Nota bene pour ceux qui ne sont pas acolytes de LF / Nota Bene for non LF-acolytes

L'habitation du typage en LF n'est malheureusement pas décidable (à cause de règles non dirigées par la syntaxe) et souvent elle demande une quantité importante d'interaction homme-machine. On ne rentrera pas dans les détails plus intimes de la forme du type dépendant σ et de $\mathtt{e}_0(\mathtt{C},\mathtt{m})$: beaucoup de développements se focalisant sur la certification d'algorithmes fonctionnels et impératifs ont été écrits dans la dernière décennie [Log07]. D'ordinaire cette activité est soumise à une annotation lourde des routines qu'on veut certifier avec un langage logique d'assertions. Des outils *ad hoc* et des pré-compilateurs manipulent ces annotations et produisent un certain nombre d'*obligations de preuves* qu'un cadre logique devrait être capable de prouver (par exemple en montrant le terme de preuve qui habite σ). On espère enrichir la prochaine version V2 de Peter avec des annotations écrites dans un langage logique d'assertions.

LF-Inhabitation is unfortunately not decidable (because of the presence of non-syntax directed typing rules) and often it requires a considerable amount of human-computer interaction. We do not want to enter into details of the shape of the dependent-type σ and $\mathtt{e}_0(\mathtt{C},\mathtt{m})$: many developments on certifying functional and imperative algorithms have been written in the last decade [Log07]. Usually this kind of activity passes through a heavy annotation of the routine we want to certify with a logic assertion language. *Ad hoc* tools and pre-compilers process annotations produce a number of *proof obligations* that a Logical Framework should be able to prove (*i.e.* by exhibition of a proof-term inhabiting σ). We plan, in the next V2 version of Peter, to enrich method bodies with annotations written in a suitable logic assertion language.

Toujours pour ceux qui ne sont pas experts de LF on présente, sans les commenter, les règles principales sous-jacentes à la *Machine à Types*

Again, for non LF-acolytes, we present, without comment, the main typing rules underneath the Bob, Furio, and Gordon's *Meta-Logical Type-*

Méta-logique de Bob, Furio et Gordon : | *Machine*:

$$\frac{\Gamma, \mathtt{x}{:}\sigma \vdash_{\mathsf{LF}} \mathtt{e} : \tau \quad \Gamma \vdash_{\mathsf{LF}} \Pi \mathtt{x}{:}\sigma.\tau : \mathsf{Type}}{\Gamma \vdash_{\mathsf{LF}} \lambda \mathtt{x}{:}\sigma.\mathtt{e} : \Pi \mathtt{x}{:}\sigma.\tau} \text{(O·Abs)}$$

$$\frac{\Gamma \vdash_{\mathsf{LF}} \mathtt{e}_1 : \Pi \mathtt{x}{:}\sigma.\tau \quad \Gamma \vdash_{\mathsf{LF}} \mathtt{e}_2 : \sigma}{\Gamma \vdash_{\mathsf{LF}} \mathtt{e}_1\, \mathtt{e}_2 : \mathtt{e}_1[\mathtt{e}_2/\mathtt{x}]} \text{(O·Appl)}$$

$$\frac{\Gamma \vdash_{\mathsf{LF}} \mathtt{e}_1 : \sigma \quad \Gamma \vdash_{\mathsf{LF}} \tau : \mathsf{Type} \quad \Gamma \vdash_{\mathsf{LF}} \sigma =_\beta \tau}{\Gamma \vdash_{\mathsf{LF}} \mathtt{e}_1 : \tau} \text{(O·Conv)}$$

$$\frac{\Gamma, \mathtt{x}{:}\sigma \vdash_{\mathsf{LF}} \tau : \mathsf{Type}}{\Gamma \vdash_{\mathsf{LF}} \Pi \mathtt{x}{:}\sigma.\tau : \mathsf{Type}} \text{(F·Pi)}$$

Table 6.6: Les principales règles de typage de LF / LF's Most Important Type Rules

6.1.7 Exemples / Examples

On présente des exemples classiques pris dans [Bar92] pour montrer la puissance de l'isomorphisme de Curry-Howard et de LF : le lecteur expert reconnaîtra aisément que le premier exemple est typé avec une discipline de type polymorphe (donc non typable dans LF) : on l'a présenté juste pour montrer la puissance du paradigme *proposition-as-types&proofs-as-terms*. Soit $\bot \equiv \Pi a{:}\mathsf{Type}.a$ qui est la définition au second ordre du *falsum*. Pour des raisons historiques, $\mathtt{A}, \mathtt{B}$ désignent des variables de type et $\mathcal{P}, \mathcal{Q}$ des constructeurs en LF.

We present some classical examples from [Bar92] showing the power of the Curry-Howard isomorphism and of the Logical Framework: the expert reader will find the first example typable in a polymorphic type-discipline (so not in LF): we do this just to show the power of the *proposition-as-types&proofs-as-terms* paradigm. Let $\bot \equiv \Pi a{:}\mathsf{Type}.a$ which is the second order definition of *falsum*. For historical reasons, $\mathtt{A}, \mathtt{B}$ denotes LF type-variables, and $\mathcal{P}, \mathcal{Q}$ for LF constructors.

- $\vdash_{\mathsf{LF}} (\lambda \mathtt{A}{:}\mathsf{Type}.\lambda \mathtt{x}{:}\bot.\mathtt{x}\,\mathtt{A}) : (\Pi \mathtt{A}{:}\mathsf{Type}.\bot \to \mathtt{A})$
  Ex falso sequitur quodlibet, *i.e.* anything follows from a false statement: the term in this type is its proof
- $\mathtt{A}{:}\mathsf{Type} \vdash_{\mathsf{LF}} \mathtt{A} \to \mathsf{Type}$
  If A is a set, then $\mathtt{A} \to \mathsf{Type}$ is a constructor on predicates on A
- $\mathtt{A}{:}\mathsf{Type}, \mathcal{P}{:}\mathtt{A} \to \mathsf{Type}, \mathtt{x}{:}\mathtt{A} \vdash_{\mathsf{LF}} \mathcal{P}(\mathtt{x}) : \mathsf{Type}$
  If A is a set, and $\mathtt{x} \in \mathtt{A}$, and $\mathcal{P}$ is a predicate on A, then $\mathcal{P}(\mathtt{A})$ is a type considered as proposition (true is inhabited; false otherwise)
- $\mathtt{A}{:}\mathsf{Type}, \mathcal{P}{:}\mathtt{A} \to \mathtt{A} \to \mathsf{Type} \vdash_{\mathsf{LF}} \Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x}\,\mathtt{x}) : \mathsf{Type}$
  If A is a set, and $\mathtt{x} \in \mathtt{A}$, and $\mathcal{P}$ is a binary predicate on A, then $\mathcal{P}(\mathtt{x}\,\mathtt{x})$ is a type considered as proposition

- $\mathtt{A}{:}\mathsf{Type}, \mathcal{P}{:}\mathtt{A} \to \mathsf{Type}, \mathcal{Q}{:}\mathtt{A} \to \mathsf{Type} \vdash_{\mathsf{LF}} \Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x}) \to \mathcal{Q}(\mathtt{x}) : \mathsf{Type}$
  This proposition states that for set-predicated $\mathcal{P}$ and $\mathcal{Q}$, the set $\mathcal{P}(\mathtt{x})$ is included in the set $\mathcal{Q}(\mathtt{x})$
- $\mathtt{A}{:}\mathsf{Type}, \mathcal{P}{:}\mathtt{A} \to \mathsf{Type} \vdash_{\mathsf{LF}} \Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x}) \to \mathcal{P}(\mathtt{x}) : \mathsf{Type}$
  This proposition states the reflexivity of the inclusion
- $\mathtt{A}{:}\mathsf{Type}, \mathcal{P}{:}\mathtt{A} \to \mathsf{Type} \vdash_{\mathsf{LF}} \lambda \mathtt{x}{:}\mathtt{A}.\lambda \mathtt{x}{:}\mathcal{P}(\mathtt{A}).\mathtt{x} : \Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x}) \to \mathcal{P}(\mathtt{x})$
  The subject of this judgment exhibits *proof* of reflexivity of the inclusion
- $$\begin{array}{rll} \mathtt{A}{:}\mathsf{Type}, \mathcal{P}{:}\mathtt{A} \to \mathsf{Type}, \mathcal{Q}{:}\mathsf{Type} & \vdash_{\mathsf{LF}} & ((\Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x}) \to \mathcal{Q}) \to \mathcal{Q}) : \mathsf{Type} \\ \mathtt{A}{:}\mathsf{Type}, \mathcal{P}{:}\mathtt{A} \to \mathsf{Type}, \mathcal{Q}{:}\mathsf{Type}, \mathtt{y}{:}\mathtt{A} & \vdash_{\mathsf{LF}} & (\lambda \mathtt{x}{:}(\Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x}) \to \mathcal{Q}).\lambda \mathtt{z}{:}(\Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x})).\mathtt{x}\,\mathtt{y}\,(\mathtt{z}\,\mathtt{y}) : \\ & & (\Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x}) \to \mathcal{Q}) \to (\Pi \mathtt{x}{:}\mathtt{A}.\mathcal{P}(\mathtt{x})) \to \mathcal{Q} \end{array}$$
  The above states that the proposition $(\forall \mathtt{x} \in \mathtt{A}.\mathcal{P}(\mathtt{x}) \to \mathcal{Q}) \to (\forall \mathtt{x} \in \mathtt{A}.\mathcal{P}(\mathtt{x})) \to \mathcal{Q}$ is true in non-empty set $\mathtt{A}$: the subject is the proof of the previous (true) proposition.

Table 6.7: Dérivations classiques en LF / LF's Classical Derivations

L'exemple suivant montre la puissance descriptive du cadre logique : les axiomes et les règles de la logique propositionnelle *à la* Hilbert et *à la* Pravitz (c-à-d. Déduction Naturelle) ; d'autres exemples peuvent être trouvés dans les premiers articles sur LF [HHP94, AHMP92].

The next example presents the descriptive power of the Logical Framework: the axioms and rules of the Propositional Logic *à la* Hilbert and *à la* Pravitz (a.k.a. Natural Deduction Style); other motivating examples can be found on the first LF's papers [HHP94, AHMP92].

Propositional Connectives

$o : \mathsf{Type} \qquad \supset : o \to o \to o \qquad \neg : o \to o$

Judgment

$\mathsf{True} \;:\; o \to \mathsf{Type}$

Axioms and Modus Ponens *à la* Hilbert

$\mathsf{A}_1 \;:\; \Pi \phi^o.\ \Pi \psi^o.\ \mathsf{True}(\phi \supset (\psi \supset \phi))$

$\mathsf{A}_2 \;:\; \Pi \phi^o.\ \Pi \psi^o.\ \Pi \theta^o.\ \mathsf{True}(\phi \supset (\psi \supset \theta) \supset (\phi \supset \psi) \supset (\phi \supset \theta))$

$\mathsf{A}_3 \;:\; \Pi \phi^o.\ \Pi \psi^o.\ \mathsf{True}((\neg \phi \supset \neg \psi) \supset (\psi \supset \phi))$

$\mathsf{MP} \;:\; \Pi \phi^o.\ \Pi \psi^o.\ \mathsf{True}((\phi \supset \psi)) \to \mathsf{True}(\phi) \to \mathsf{True}(\psi)$

Natural Deduction Styles Rules *à la* Prawitz

$\neg\mathsf{I} \;:\; \Pi \phi^o.\ \Pi \psi^o.\ (\mathsf{True}(\phi) \to \mathsf{True}(\psi)) \to (\mathsf{True}(\phi) \to \mathsf{True}(\neg \psi)) \to \mathsf{True}(\neg \phi)$

$\neg\mathsf{E} \;:\; \Pi \phi^o.\ \mathsf{True}(\neg\neg \phi) \to \mathsf{True}(\phi)$

$\supset\mathsf{I} \;:\; \Pi \phi^o.\ \Pi \psi^o.\ (\mathsf{True}(\phi) \to \mathsf{True}(\psi)) \to \mathsf{True}(\phi \supset \psi)$

$\supset\mathsf{E} \;:\; \Pi \phi^o.\ \Pi \psi^o.\ \mathsf{True}(\phi \supset \psi) \to \mathsf{True}(\phi) \to \mathsf{True}(\psi)$

Table 6.8: Logique propositionnelle *à la* Hilbert / Propositional Logic *à la* Hilbert

6.2 Une généralisation du cadre logique / A Generalization of the LF

Même si LF permet (plutôt bien) d'encoder des règles logiques comme des fonctions de preuves vers des preuves, il reste peu expressif pour ce qui concerne les «*side conditions*» qui peuvent renforcer l'application d'une règle. Puisque l'application d'une règle est encodée en LF comme une β-réduction, il n'existe pas beaucoup de moyens pour encoder des clauses conditionnelles, même les plus simples, comme celles qui apparaissent dans des *règles de preuve*. On rappelle qu'une règle de preuve peut être appliquée seulement à des prémisses qui ne dépendent d'aucune hypothèse ; cela à l'opposé des *règles de dérivation* qui peuvent être appliquées partout. Les règles faisant partie de «logiques modales» et de «logiques des programmes» présentées en déduction naturelle et peuvent être problématiques à exprimer en LF.

Although LF, very rightly so, allows us to encode rules as functions from proofs to proofs, it is nevertheless a little restrictive in defining the "side conditions" that it can enforce on the application of rules. Rule application being encoded simply as lambda application, there are only roundabout ways to encode provisos, even as simple as the ones appearing in a *rule of proof*. Recall that a rule of proof can be applied only to premises which do not depend on any assumption, as opposed to a *rule of derivation* which can be applied everywhere. Also rules which appear in many natural deduction presentations of modal and program logics are very problematic in standard LF.

Beaucoup de ces systèmes possèdent des règles qui peuvent être appliquées seulement à des prémisses qui dépendent de certaines hypothèses d'une forme particulière [CH84] ou à des dérivations qui peuvent être construites seulement à partir d'une séquence précise de règles. Pour terminer, les Logiques Linéaire ou Relevant semblent être encodables dans LF avec beaucoup d'*overhead*. Dans le passé, plusieurs extensions de LF ont été proposées. Souvent, le prix à payer est une complication excessive du cadre logique. Par contre, le *desideratum* a toujours été celui d'avoir un cadre logique composé d'un *télescope* de systèmes, chacun étant une extension conservatrice de l'autre, qui permet d'encoder, par accroissement, des *side-conditions* de plus en plus méchantes. C'est précisément ce qu'on propose dans l'extension du métalangage de Peter.

Many such systems feature rules which can be applied only to premises which depend solely on assumptions of a particular shape [CH84], or whose derivations have been carried out using only certain sequences of rules. Finally, Linear or Relevance Logics appear to be encodable only using a very heavy machinery. In the past, extensions of LF have often been proposed. The complexity price to pay, however, was always very high as far as the language theory. The *desideratum* has always been that of having a metalogical framework, *i.e.* a *telescope* of systems, each a conservative extension of the previous ones, which can incrementally and naturally encode nastier and nastier classes of side-conditions. This is precisely what we propose in this extension of Peter's meta-logical language.

L'idée est plutôt simple. Elle consiste à enlever un «*bandeau sur les yeux*», en faisant bien la différence entre deux concepts différents qui se sont effondrés en un seul dans la définition originale de LF, c-à-d. qu'ils sont considérés égaux par définition. Comme déjà dit, la rigidité de LF vient du fait que la β-réduction s'applique avec trop de liberté. On voudrait restreindre cette application, mais le système de types ne semble pas assez riche pour pouvoir exprimer cette restriction. Ce qu'on propose est d'utiliser comme type de l'application, dans la nouvelle règle d'application (O·Appl·New), non pas le type obtenu en substituant des termes

The key idea is extremely simple. It amounts to remove a *"blind spot"*, thus making explicit the differences between two different notions, which are conflated to only one, in the original LF, *i.e.* which are taken to be definitionally equal. As already mentioned, much of the rigidity of LF arises from the fact that β-reduction can be applied too generally. One would like to restrict it, but the type system appears not to be rich enough to be able to express such restrictions. What we propose is to use as type of an application, in the new term application rule (O·Appl·New) below, not the type which is obtained by carrying out directly in the met-

du métalangage dans le type dépendant, mais un nouveau type qui enregistre simplement l'information selon laquelle une telle réduction doit être faite ou non. Avec une application de la règle de conversion on pourra récupérer le comportement original de LF. L'ancienne (O·Appl) et la nouvelle règle (O·Appl·New) de typage sont données ci-dessous :

alanguage the substitution of the argument in the type, but a new form of type which simply records the information that such a reduction needs to be carried out. An application of the Type Conversion Rule can then recover the usual effect of the application rule. The old typing-rule (O·Appl) rule and the new rule (O·Appl·New) appear as follows:

$$\frac{\Gamma \vdash_{\mathsf{LF}} \mathsf{e}_1 : \Pi x{:}\sigma.\tau \quad \Gamma \vdash \mathsf{e}_2 : \sigma}{\Gamma \vdash_{\mathsf{LF}} \mathsf{e}_1\,\mathsf{e}_2 : \tau[\mathsf{e}_2/x]}\ (\text{O·Appl}) \qquad \frac{\Gamma \vdash_{\mathsf{PLF}} \mathsf{e}_1 : \Pi x{:}\sigma.\tau \quad \Gamma \vdash_{\mathsf{PLF}} \mathsf{e}_2 : \sigma}{\Gamma \vdash_{\mathsf{PLF}} \mathsf{e}_1\,\mathsf{e}_2 : (\lambda x{:}\sigma.\tau)\,\mathsf{e}_2}\ (\text{O·Appl·New})$$

Table 6.9: La vieille et la nouvelle règle d'application pour le cadre logique / The Old and the New Rule for the Logical Framework

Souvent on dit : «*le mieux est l'ennemi du bien*». Et une fois qu'on a choisi cette stratégie, on a un moyen pour annoter dans les types l'information selon laquelle une réduction peut (ou pas) être effectuée dans le terme (ou non). En d'autres termes, le type n'est pas *égal par définition* au type réduit par conversion, comme dans le cas de LF. Sans modifier beaucoup la règle de typage, on a une façon naturelle de typer des calculs qui généralisent ou restreignent l'application de la β-réduction en attendant qu'une contrainte sur le métalangage soit satisfaite ou pas. Chacun de ces calculs pourrait être considéré comme un candidat potentiel pour un nouveau cadre logique, où la complexité ajoutée dans les termes pourrait être naturellement apprivoisée en utilisant la puissance d'expression du nouveau système de types. Une fois que cette extension a été fixée d'une façon modulaire, on a le cadre logique télescopique qu'on cherchait.

Pour procéder avec un maximum de généralité, on introduit un nouveau type de λ- et Π-abstractions, $\lambda\mathcal{P}{:}\Delta.\tau$ et $\Pi\mathcal{P}{:}\Delta.\mathsf{e}$. Le prédicat unaire $\mathcal{P}$ est complètement général et le contexte de type $\Delta \triangleq [x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n]$ représente les variables liées par Π et λ dans $\Pi\mathcal{P}{:}\Delta.\tau$ et $\lambda\mathcal{P}{:}\Delta.\mathsf{e}$. Ce prédicat pourra être «instancié» de différentes façons. Par exemple, il pourrait renforcer le fait que l'argument est clos ou que toutes les variables libres ont un type d'une forme précise. Cela nous permettrait de récupérer calculs et cadres, comme LF lui-même, le Calcul de Réécriture [29,30] et le lambda calcul par valeurs de

As it is often said: sometimes, *"less is more"*. Once this move has been made, we have a mechanism for annotating in a type the information that a reduction is waiting to be carried out in the term. Taking this move seriously the type of a term need not be necessarily *definitionally equal* to the reduced type. We can generalize further our approach, without much hassle: we have a principled and natural way of typing calculi featuring generalized or restricted forms of β-reduction which wait for some constraint to be satisfied before reducing. Each such calculus can be considered as a potential candidate for underpinning a new Logical Framework, where all the extra complexity in terms can be naturally tamed utilizing the expressive power of the new typing system. Once this extension is carried out in a sufficiently modular form, we obtain the telescopic metalogical framework we were looking for.

In order to proceed in full generality we introduce a new form of λ and corresponding Π abstraction $\lambda\mathcal{P}{:}\Delta.\tau$ and $\Pi\mathcal{P}{:}\Delta.\mathsf{e}$. The unary predicate $\mathcal{P}$ is completely general at this stage, and the type context $\Delta \triangleq [x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n]$ denotes the variables bound by Π and λ in $\Pi\mathcal{P}{:}\Delta.\tau$ and $\lambda\mathcal{P}{:}\Delta.\mathsf{e}$. The predicate can be instantiated in various useful ways. For instance, it can enforce the fact that the argument is closed, or that all its free variables have a type of a given form. This format can also recover many existing calculi in the literature such as LF, the Rewriting Calculus [29,30], and the Plotkin's call-by-value lambda calculus [Plo75].

Plotkin [Plo75]. Dans tous les cas, l'application d'égalité de types sera utilisée pour récupérer, d'une façon *conservatrice* le comportement d'une β-réduction :

$$(\lambda\mathcal{P}{:}\Delta.\mathtt{e}_1)\,\mathtt{e}_2 \longrightarrow \mathtt{e}_1\widehat{\mathcal{P}}(\mathtt{e}_2)$$

si $\mathcal{P}(\mathtt{e}_2)$ est valide et $\widehat{\mathcal{P}}(\mathtt{e}_2)$ est une substitution. Les types provenant des cas où le prédicat $\mathcal{P}$ est faux identifient, avec précision, l'endroit où l'application n'a pas pu se déclencher avec succès. On peut immédiatement comprendre comment les «règles de preuve» peuvent être exprimées facilement en restreignant leurs applications à des termes clos.

Cette idée de distinguer entre ces deux notions qui étaient aplaties en une seule représente un petit pas pour un système de types mais un pas important dans un cadre logique. L'idée de capitaliser les similarités entre les opérateurs «λ» et «Π» n'est pas nouvelle, voir *e.g.* [dB80, KBN99], mais la nouveauté est de la «capitaliser» avec les idées nouvelles du Calcul de Réécriture typé présenté par Horatiu, Claude et moi-même dans l'article *The Rho Cube* [29]. En faisant cela, on conçoit une classe de lambda calculs typés avec motifs et beaucoup plus...

In all cases, an application of the type equality rule can be used to recover, *conservatively*, the effect of successful β-reductions:

$$(\lambda\mathcal{P}{:}\Delta.\mathtt{e}_1)\,\mathtt{e}_2 \longrightarrow \mathtt{e}_1\widehat{\mathcal{P}}(\mathtt{e}_2)$$

if $\mathcal{P}(\mathtt{e}_2)$ holds and $\widehat{\mathcal{P}}(\mathtt{e}_2)$ is a substitution. The extra types deriving from failures allow for precisely the extra elbow-room that is needed to prevent the applications of certain rules too loosely. It is now immediate to see that "rules of proof" can be dealt with straightforwardly by restricting applications to closed terms.

This idea of distinguishing between two notions which were previously flattened into one is a small step for a type system but a momentous step for a Logical Framework. The idea of capitalizing on the similarities between the "λ" and "Π" operators is not new, see *e.g.* [dB80, KBN99], but what we do here is to capitalize on it, in the type system, as was done in the work by Horatiu, Claude and myself in the *The Rho Cube* [29]. By so doing, we allow for a generalized form of pattern lambda calculi, and also go beyond...

6.2.1 Syntaxe / Syntax

La digression qu'on vient de faire, inspirée par [6], sort un peu de l'extension du métalangage de Sage Peter qu'on avait en tête (pour le dire en termes musicaux excusez la «*fuga*»). Pour l'extension on se restreint à une «instanciation» du cadre logique générique GLF, appelé *cadre logique avec motifs*, PLF, où on considère qu'une instanciation de $\lambda\mathcal{P}{:}\Delta.e$ comme $\lambda\mathrm{Match}(\mathtt{alg}(\overline{\mathtt{e}}),\mathtt{y}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).\mathtt{e}$, où Match est un algorithme de filtrage du premier ordre *à la* Huet [Hue76], $\mathtt{alg}(\overline{\mathtt{e}})$ est un terme algébrique comme on vient de le voir dans le Chapitre 5 sur les îles, $\mathtt{y}$ est le paramètre formel lié à λ et $\overline{\mathtt{x}}{:}\overline{\mathtt{C}}$ sont les variables libres de $\overline{\mathtt{e}}$ avec ses types. Formellement le prédicat binaire Match est défini comme suit :

The above digression, taken from [6], goes far away to the extension of Peter's logical language we have in mind (in "musical" jargon we would say: sorry for the *fuga*). For this further extension, we restrict to a suitable instantiation of the General Logical Framework, called *Pattern Logical Framework*, PLF, that is we consider a suitable instantiation of $\lambda\mathcal{P}{:}\Delta.e$ as $\lambda\mathrm{Match}(\mathtt{alg}(\overline{\mathtt{e}}),\mathtt{x}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).e$, where $\mathrm{Match}(\_,\_)$ is a first-order matching algorithm *à la* Huet [Hue76], $\mathtt{alg}(\overline{\mathtt{e}})$ is an algebraic term, $\mathtt{x}$ is the λ-bound variable, and $\overline{\mathtt{x}}{:}\overline{\mathtt{C}}$ are the free-variables of $\overline{\mathtt{e}}$ with related types. As such, our PLF feature a general form of pattern β-reduction. Formally, the binary predicate Match is defined as follows:

$$\mathrm{Match}(\mathtt{alg}(\overline{\mathtt{e}});\mathtt{e}) \triangleq \begin{cases} \mathit{true} & \text{if } \exists\theta.\ \mathtt{alg}(\overline{\mathtt{e}})\theta \equiv \mathtt{e} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Table 6.10: La fonction binaire Match adoptée dans PLF / PLF's Binary Function Match

Par souci de simplicité et avec un petit abus de notation par rapport à la formulation originelle de [6], on utilisera $\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}})$ plutôt que $\mathrm{Match}(\mathtt{alg}(\overline{\mathtt{e}}),\mathtt{x})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}})$ dans les abstractions et dans les types produits : notre extension de syntaxe sera donc :

For the sake of simplicity, and with a little abuse of notation w.r.t. the original formulation of [6], we will use in abstractions and product-types $\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}})$ instead of $\mathrm{Match}(\mathtt{alg}(\overline{\mathtt{e}}),\mathtt{x})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}})$: our syntax extension is as follows:

σ	::=	$\Pi\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\sigma \mid \lambda\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\sigma$	PLF−types (Propositions)
K	::=	$\Pi\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\mathtt{K} \mid \lambda\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\mathtt{K}$	PLF−kinds (Constructors)
e	::=	$\mathtt{alg}(\overline{\mathtt{e}}) \mid \lambda\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\mathtt{e}$	PLF−terms (Proofs)

Table 6.11: La syntaxe du cadre logique avec motifs / Pattern Logical Framework's Syntax

6.2.2 Règles clés pour l'exécution / Key Run-time Rules

Même si les termes PLF ne doivent pas être exécutés, la présence de types dépendants oblige, dans le processus d'habitation d'un type, à l'utilisation d'une règle de conversion pour les types qui les «exécute» dans leur forme normale. On présente les nouvelles règles de la sémantique opérationnelle, où $\theta = \mathrm{Match}(\mathtt{alg}(\overline{\mathtt{e}});\mathtt{e})$ est le résultat d'une exécution d'un algorithme de filtrage du premier ordre.

Even if PLF-terms need not to be executed, the presence of dependent-types forces type inhabitation to make use of a conversion rule that "runs" types to their normal form. The new operational semantics rules are described below, where $\theta = \mathrm{Match}(\mathtt{alg}(\overline{\mathtt{e}});\mathtt{e})$ is the result of a run of a first-order pattern matching algorithm.

($\beta_{\mathtt{alg}}$−Objects)	$(\lambda\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\mathtt{e}_0)\,\mathtt{e}$	$\mapsto_\beta$	$\mathtt{e}_0\theta$
($\beta_{\mathtt{alg}}$−Families)	$(\lambda\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\sigma)\,\mathtt{e}$	$\mapsto_\beta$	$\sigma\theta$
($\beta_{\mathtt{alg}}$−Kinds)	$(\lambda\mathtt{alg}(\overline{\mathtt{e}})\!:\!(\overline{\mathtt{x}}\!:\!\overline{\mathtt{C}}).\mathtt{K})\,\mathtt{e}$	$\mapsto_\beta$	$\mathtt{K}\theta$

Table 6.12: Règles d'exécution à petit pas du cadre logique avec motifs / Pattern Logical Framework's Small-step Execution Rules

6.2.3 Règles clés pour le typage / Key Type-checking Rules

Les règles principales qui prennent en considération l'extension du cadre logique avec des motifs sont les suivantes :

The main typing rules taking into account the extension of the Logical Framework with patterns are the following ones:

$$\frac{\Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{C}} \vdash_{\mathsf{PLF}} \mathtt{alg}(\overline{\mathtt{e}}) : \sigma \quad \Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{C}} \vdash_{\mathsf{PLF}} \mathtt{e} : \tau}{\Gamma \vdash_{\mathsf{PLF}} \lambda\mathtt{alg}(\overline{\mathtt{e}}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).\mathtt{e} : \Pi\mathtt{alg}(\overline{\mathtt{e}}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).\tau}\ (\text{O·Abs})$$

$$\frac{\Gamma \vdash_{\mathsf{PLF}} \mathtt{e}_1 : \Pi\mathtt{alg}(\overline{\mathtt{e}}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).\tau \quad \Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{C}} \vdash_{\mathsf{PLF}} \mathtt{alg}(\overline{\mathtt{e}}) : \sigma \quad \Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{C}} \vdash_{\mathsf{PLF}} \mathtt{e}_2 : \sigma}{\Gamma \vdash_{\mathsf{PLF}} \mathtt{e}_1\,\mathtt{e}_2 : (\lambda\mathtt{alg}(\overline{\mathtt{e}}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).\tau)\,\mathtt{e}_2}\ (\text{O·Appl})$$

Table 6.13: Les principales règles de typage du cadre logique avec motifs / Pattern Logical Framework's Main Typing Rules

Dans une «coquille de noix» :
• (O·Abs) Cette règle d'abord vérifie d'abord le bon typage des motifs à filtrer (voir l'analogie forte avec les cape des îles), puis type le corps de l'abstraction et, enfin, construit un type dépendant où le motif algébrique est enregistré ;
• (O·Appl) Cette règle d'abord vérifie d'abord le type de la fonction avec un type dépendant $\Pi\mathtt{alg}(\overline{\mathtt{e}}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).\tau$, puis type le motif (ou filtre algébrique) dans un environnement de type enrichi des types des variables libres de $\overline{\mathtt{e}}$; ensuite, elle type l'argument qui doit être du même type que le filtre algébrique et, enfin, construit un type dépendant (pas en forme normale) obtenu en «ouvrant le télescope» présenté dans le cadre logique général GLF. Grâce à la propriété de Church-Rosser, la règle de conversion sur les types s'occupera de réduire ce type dépendant en sa forme-normale.

In a nutshell:
• (O·Abs) This rule first type-checks the pattern to be matched (observe the strong analogy with island's cape), then it type-checks the body of the abstraction, and finally it builds a dependent-type where the algebraic pattern is recorded;
• (O·Appl) This rule first type-checks the function with a dependent-type $\Pi\mathtt{alg}(\overline{\mathtt{e}}){:}(\overline{\mathtt{x}}{:}\overline{\mathtt{C}}).\tau$, then it type-checks the pattern (or algebraic filter) in an environment enriched with the type of the free variables of $\overline{\mathtt{e}}$, then it type-checks the argument that must be of the same type of the pattern, and finally it builds a dependent-type (not in normal-form) obtained by instrumenting the "telescope" introduced in the General Logical Framework GLF. Thanks to the Church-Rosser property, a conversion rule will run this type to its normal-form.

6.2.4 Exemples / Examples

On commence en montrant des codages en PLF : après on montrera des exemples dans Sage Peter. Comme premier exemple, on encode en PLF la version basique du Calcul de Réécriture non typé ; le lecteur observera que l'algorithme de filtrage Match(_, _) est exactement celui du Calcul de Réécriture non typé. Ceci simplifiera énormément l'encodage car tous les détails concernant l'algorithme de filtrage dans le langage à encoder seront traités directement dans le métalangage lui-même.

We start by first showing some PLF encoding: then we proceed to show genuine examples in Wise Peter. As a first example, we encode in PLF the simplest basic version of the Untyped Rewriting Calculus; note that the matching algorithm Match(_, _) is exactly the same both in the metalanguage and in the language to be encoded. This greatly simplifies the encoding because it pushes in the metalanguage the filtering issues of the language to be encoded.

Syntax and Operational Semantics

$$\begin{array}{llll} Q & ::= & f \mid Q\,P & (P \to M)\,N \quad \to_\rho \quad M\theta \quad \text{with } \theta = \mathrm{Match}(P;N) \\ P & ::= & x \mid f \mid Q\,P & (M_1\,,M_2)\,M_3 \quad \to_\delta \quad (M_1\,M_3\,,M_2\,M_3) \\ M & ::= & x \mid P \to M \mid M\,M \mid M\,,M & \text{N.B. both } (M_1\,,M_2) \text{ and } \to_\delta \text{ are derivable} \end{array}$$

Syntactic Categories Operations and Judgments ($o^n \triangleq \overbrace{o \to \ldots \to o}^{n \text{ times}}$ and $\checkmark\mathsf{C}[\overline{x}]{:}(\overline{x{:}o^n}) \triangleq \checkmark\mathsf{C}[\overline{x^{o^n}}]$)

o : Type Alg : o^2 Rew : $o^2 \to o$ App : o^3 Pair : o^3 $=$: $o \to o \to$ Type

Rewriting encoding $[\![-]\!]$: Rho $\Rightarrow$ PLF

$$\begin{array}{llllll} [\![x]\!] & \triangleq & x & [\![P \to M]\!] & \triangleq & \mathsf{Rew}\ (\lambda [\![P]\!]{:}\Delta.\ [\![M]\!]) \quad \Delta \triangleq \overline{\mathsf{Fv}(P){:}o} \\ [\![f]\!] & \triangleq & \mathsf{Alg}\ f & [\![M\,N]\!] & \triangleq & \mathsf{App}\ [\![M]\!]\,[\![N]\!] \\ [\![Q\,P]\!] & \triangleq & \mathsf{App}\ [\![Q]\!]\,[\![P]\!] & [\![M\,,N]\!] & \triangleq & \mathsf{Rew}\ (\lambda x{:}o.\ \mathsf{Pair}\ (\mathsf{App}\ [\![M]\!]\,x)\,(\mathsf{App}\ [\![N]\!]\,x)) \end{array}$$

Axioms and Rules

$\mathsf{Eq}_{\mathsf{refl}}$ $\mathsf{Eq}_{\mathsf{symm}}$ $\mathsf{Eq}_{\mathsf{trans}}$ $\mathsf{Eq}_{\mathsf{ctx}}$ as in the lambda calculus encoding (see [HHP94])

Rho : $\Pi r{:}o^2.\ \Pi a{:}o.\ \mathsf{App}\ (\mathsf{Rew}\ r)\,a = r\,a$

Eta : $\Pi x{:}o.\ \mathsf{Rew}\ (\lambda y{:}o.\ \mathsf{App}\ x\,y) = x$

Xi : $\Pi r{:}o^2.\ \Pi s{:}o^2.\ (\Pi a{:}o.\ r\,a = s\,a) \to \mathsf{Rew}\ r = \mathsf{Rew}\ s$

Delta : $\Pi\mathsf{Rew}\ (\lambda x{:}o.\ \mathsf{Pair}\ (\mathsf{App}\ y^o\,x)\,(\mathsf{App}\ z^o\,x)).\ \Pi a{:}o.$
$\mathsf{App}\ (\mathsf{Rew}\ (\lambda x{:}o.\ \mathsf{Pair}\ (\mathsf{App}\ y\,x)\,(\mathsf{App}\ z\,x)))\,a =$
$\mathsf{Rew}\ (\lambda v{:}o.\ \mathsf{Pair}\ (\mathsf{App}\ (\mathsf{App}\ y\,a)\,v)\,(\mathsf{App}\ (\mathsf{App}\ z\,a)\,v))$

Table 6.14: Encodage du calcul de réécriture en PLF / Encoding of the Rewriting Calculus in PLF

Comme deuxième exemple, on montre comment le lambda calcul avec appel par valeurs de Plotkin peut être encodé en PLF; l'utilisation du motif $!\,(x)$ dans le typage de la constante Betav permet d'éviter l'introduction d'une sous-catégorie syntaxique pour o.

As a second example, we illustrate how Plotkin's call-by-value lambda calculus can be encoded in PLF; observe the crucial use of the algebraic pattern $!\,(x)$ in the typing of the constant Betav that avoids the introduction of o's subcategories.

Operational Semantics

$(\lambda x.M)\,N \to_{\beta_\mathsf{v}} M[N/x]$ if N is a value

Syntactic Categories, Constructors and Judgments

o : Type ! : o^2 Lam : $\Pi f{:}[\Pi!\,(x^o).o].\ o$ App : o^3 $=$: $o \to o \to$ Type

Axioms and Rules

$\mathsf{Eq}_{\mathsf{refl}}$: $\Pi x^o.\ x = x$

$\mathsf{Eq}_{\mathsf{symm}}$: $\Pi x^o.\ \Pi y^o.\ (x = y) \to (y = x)$

$\mathsf{Eq}_{\mathsf{trans}}$: $\Pi x^o.\ \Pi y^o.\ \Pi z^o.\ (x = y) \to (y = z) \to (x = z)$

$\mathsf{Eq}_{\mathsf{ctx}}$: $\Pi x^o.\ \Pi y^o.\ \Pi z^o.\ \Pi w^o.\ (x = y) \to (z = w) \to (\mathsf{App}\ x\,z = \mathsf{App}\ y\,w)$

Betav : $\Pi f{:}[\Pi!\,(x^o).o].\ \Pi y^o.\ \mathsf{App}\ !\,(\mathsf{Lam}\ f)\,!\,(y) = f\,!\,(y)$

Xiv : $\Pi f{:}[\Pi!\,(x^o).o].\ \Pi g{:}[\Pi!\,(x^o).o].\ (\Pi z^o.\ f\,!\,(z) = g\,!\,(z) \to !\,(\mathsf{Lam}\ f) = !\,(\mathsf{Lam}\ g))$

Etav : $\Pi x^o.\ !\,(\mathsf{Lam}\ (\lambda!\,(y^o).\mathsf{App}\ !\,(x)\,!\,(y))) = !\,(x)$

Table 6.15: Lambda calcul par valeurs en PLF/ Call-by-Value Lambda Calculus in PLF.

Le troisième et le quatrième examples montrent comment mélanger îles et preuves dans Sage Peter : cet exemple présente une petite classe avec son île et son certificat :

The third and the fourth examples show how to mix island and proofs in Wise Peter: the example shows a simple class with its island and its certificate:

```
cert C is {σ;e}
class Safe_Island extends Object withcert C
{;super();
 string mp(string,string name) {return man(name) -> mortal(name);} withcert C}
```

where the ''logic'' proposition φ *is* $\forall x.\ man(x) \implies mortal(x)$

encoded as the PLF*-type* σ *(depending on* `Safe_Island`, `mp`, *and the constructor* `True`:$o\rightarrow$Type*)*

`Πthis:Safe_Island.Πthis.mp(man(name)):(name:string).True(mortal(name))`

inhabited by the PLF*-term* `e`*:*

`λthis:Safe_Island.λthis.mp(man(name)):(name:string).mortal(name)`

whose ''squeezed'' computational content is: `λman(name).mortal(name)`

that is the real island-code: `man(name) -> mortal(name)`

Table 6.16: Mélanger des îles et des certificats dans Sage Peter / Mix Islands and Certificates in Wise Peter

Le dernier exemple présente un *raffinement des programmes* dans le cas simple de la méthode factorielle. Le lecteur appréciera ici le fait qu'une île peut fournir des résultats multiples (caractéristique héritée directement du Calcul de Réécriture). On commence par montrer deux versions de la fonction factorielle (une récursive et l'autre itérative) qui n'utilisent pas le prédécesseur[1], qui n'est pas primitif.

The last example shows a *program refinement* in case of a simple factorial method. The reader can appreciate the fact that one island can produce multiple results (feature which is directly inherited by the Rewriting Calculus). We show first two mathematics definition of the factorial function, a recursive and an iterative one without making use of the predecessor[2], since not primitive.

1 Cela simplifie les démonstrations.

2 This will simpliy proofs.

$$\begin{array}{llll} f\_rec(1) & \rightarrow & 1 & f\_ite(1,m) \rightarrow m \\ f\_rec(n+1) & \rightarrow & n * f\_rec(n) & f\_ite(n+1,m) \rightarrow f\_ite(n, n*m) \end{array}$$

```
cert C is {σ;}
class Last_Island extends Object withcert C
{;super();
int⁶ refine(int,int n int m){return fact(n,m)    -> this.refine(f_rec(n)),
                                    fact(n,m)    -> this.refine(f_ite(n,m)),
                                    f_rec(1)     -> 1,
                                    f_rec(n+1)   -> n*this.refine(f_rec(n)),
                                    f_ite(1,m)   -> m,
                                    f_ite(n+1,m) -> this.refine(f_ite(n,n*m));}
                              withcert C}

Last_Island i = new(Last_Island());
            i.refine(fact(4,1))        two fact cape filter and the result will be a pair
            -> <24,24>                                of type int² supertype of int⁶
```

Table 6.17: La dernière île / The Last Island

où la proposition logique et le PLF-certificat en Sage Peter sont :

where the logic proposition and the PLF's certificate in Wise Peter are:

where the ''logic'' proposition φ *is encoded as the* PLF*-type* σ	$\forall n.\ f\_rec(n) = f\_ite(n,1)$
Πthis:Last_Island.Πthis.refine(fact(n,1)):(n:int). π_1(this.refine(fact(n,1)) = π_2(this.refine(fact(n,1))	
depending of Last_Island, refine, *the constructor* =: $o \to o \to$ Type, *and the projections* $\pi_{1,2}$ *and inhabited by the* PLF*-term :...left for the ''next'' future user-friendly proof assistant...*	

Table 6.18: Le dernier certificat / The Last Certificate

La correction entre les fonctions mathématiques factorielles et la toute première intuition de factorielle qu'on avait en tête doit rester «sur papier» car elle est pré-formelle.

The correctness between the mathematical factorial functions and the very first intuition we have in mind of factorial must be "on paper" since it is pre-formal.

Sources d'inspiration / Inspiration Sources

Ce chapitre est le résultat d'une dizaine d'années d'études dans la théorie des types, métalangages et cadres logiques pour les assistants de preuves et dans les langages objets. Concernant les articles les plus théoriques que j'ai écrits au cours de la dernière dizaine d'années, ce chapitre a été influencé par mon travail de doctorat [74] et articles corrélés [11,43], par un certain nombre d'articles écrits avec Claude et Horatiu, au sein des projets Miró et Protheo à Nancy, pour adapter les types dépendants au calcul de réécriture ([16,26,29] juste pour citer les plus importants), par mes premières expériences d'encodage de calculs à objets purs dans un cadre logique avec Bernard, Marino et Alberto [7,23,25] et par les dernières recherches dans la théorie des types dépendants «épicés» avec des aspects relatifs au filtrage, avec Furio, Marina et Ivan [6,49,63,64].

This chapter is the outcome of a decade of studying type theories for proof assistants and for object-oriented programming languages. Concerning theoretical papers on dependent-type theory I have written in the last decade, this chapter was strongly influenced by my Ph.D. work [74] and subsequent papers [11,43], by a certain number of papers written with Claude and Horatiu, within the Miró and Protheo Project-teams in Nancy adapting dependent-types to the Rewriting Calculus ([16,26,29], just to cite the more important ones), by the first experiments of formalizing object-based languages in a Logical Framework with Bernard, Marino, and Alberto [7,23,25], and by the latest advances in dependent-type theory spiced with pattern-matching features, with Furio, Marina, and Ivan [6,49,63,64].

Appendix A

Peter V1: Syntaxe et sémantique / Peter V1: Syntax and Semantics

A.1 Syntaxe / Syntax

CL	::=	class C extends D [imports $\overline{\text{TA}}$] {$\overline{\text{I}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$} [with cert C]	Class Declarations
$\mathcal{CL}$	::=	cert C {σ; [e]} | cert C extends D {$\overline{\text{C}}$ $\overline{\text{m}}$}	Certificate Declaration
TL	::=	trait T [imports $\overline{\text{TA}}$] {$\overline{\text{M}}$; [$\overline{\text{S}}$]}	Trait Declarations
TA	::=	T | TA with {m@n} | TA minus {m}	Trait Alterations
I	::=	C | T | Mytype	Types
K	::=	C($\overline{\text{I}}$ $\overline{\text{f}}$){super($\overline{\text{f}}$); this.$\overline{\text{f}}$ = $\overline{\text{f}}$;}	Constructors
S	::=	I m($\overline{\text{I}}$ $\overline{\text{x}}$);	Signatures
M	::=	I m($\overline{\text{I}}$ $\overline{\text{x}}$){return e;} [with cert C]	Plain Methods
		C m($\overline{\text{C}}$ $\overline{\text{x}}$){modify {$\overline{\text{C}}$ $\overline{\text{f}}$; $\overline{\text{M}}$} and {return e;}}}	Self−Mods. Methods
		$\overline{\text{C}}$ m(E, $\overline{\text{C}}$ $\overline{\text{x}}$, $\overline{\text{D}}$ $\overline{\text{alg}}$){return $\overline{\text{e}}$;}	Island Methods
e	::=	x | e.f | e.m($\overline{\text{e}}$) | new C($\overline{\text{e}}$) | (I)e |	Basic Expressions
		usetype {e} in {e}	Pluggable Type Idiom
		p -> e | alg($\overline{\text{e}}$)	Island′s Idioms
σ	::=	a | Pi(p:δ).σ | fun(p:δ).σ | σ(e)	PLF−Propositions
K	::=	Type | Pi(p:δ).K | fun(p:δ).K | K(e)	PLF−Constructors
e	::=	p | fun(p:δ).e | e(e)	PLF−Proofs
p	::=	x | alg($\overline{\text{e}}$)	PLF−Patterns
δ	::=	σ | ($\overline{\text{x}}$:$\overline{\text{I}}$)	PLF−Decorations
$\{\vdash_{\text{norm}}, \{\vdash_{\text{ele}} \sqsubseteq \vdash_{\text{philo}}\}, \vdash_{\text{joung}}, \vdash_{\text{ado}}, \vdash_{\text{fly}}, \{\vdash_{\text{PLF}} \sqsubseteq \vdash_{\text{LF}}\}, \vdash_{\text{sharp}}\} \sqsubseteq \vdash_{\text{baby}}$			$\sqsubseteq$−Compatibility

Table A.1: La syntaxe de Peter V1 / The Peter V1 Syntax

Appendix B

Une sélection des meilleurs articles de Luigi / A Selection of the Best Luigi's Papers

J'ai choisi de mettre en appendice les articles suivantes :

I have chose to put in appendix the following papers:

(1) L. Liquori, A Spiwack. FeatherTrait: A Modest Extension of Featherweight Java.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/toplas-07.pdf

(2) L. Liquori, A Spiwack. Extending FeatherTrait Java with Interfaces.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/tcs-draft-07.pdf

(4) L. Liquori, B. Serpette. iRho: An Imperative Rewriting-calculus.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/iRho.pdf

(6) F. Honsell, M. Lenisa, L. Liquori. A Framework for Defining Logical Frameworks.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/Plotkin.pdf

(26) G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/popl-03.ps.gz

(29) H. Cirstea and C. Kirchner and L. Liquori. The Rho Cube.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/fossacs-01.ps.gz

(30) H. Cirstea and C. Kirchner and L. Liquori. Matching Power.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/rta-01.ps.gz

(34) L. Liquori. Bounded Polymorphism for Extensible Objects.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/types-98.ps.gz

(35) P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-Inflicted Extension.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/oopsla-98.ps.gz

(36) L. Liquori. On Object Extension.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/ecoop-98.ps.gz

(37) L. Liquori. An Extended Theory of Primitive Objects: First Order System.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/ecoop-97.ps.gz

(11) S. van Bakel, L. Liquori, S. Ronchi della Rocca and P. Urzyczyn. Comparing Cubes of Typed and Type Assignment Systems.
http://www-sop.inria.fr/mascotte/Luigi.Liquori/PAPERS/apal-97.ps.gz

B.1 Tout dans une page / One Page Fits All

Cursus vitæ / Cursus vitæ

Diplomas/Studies/Positions	**Year**	**Where**
Full Time Researcher (*CR1*, rank:1st)	2001- ——	INRIA Sophia-Antipolis, Lorraine, France
Lecturer (*MdC 1C*)	1999-2001	École Nat. Sup. des Mines de Nancy (ENSMN), France
Temporary Lecturer (ATER)	1998-1999	École Normale Supérieure de Lyon (ENSL), France
Post-Doctorate	1997-1998	University of Udine, Italy
Software Engineer	1996-1997	CSELT (now Telecom Italia Labs), Turin, Italy
Mathematical and Computer Science teacher	1995-1996	Scientific college *Sommelier*, Turin, Italy
Ph.D. in Computer Science	1992-1996	University of Turin, Italy
Mathematical teacher	1991-1992	Scientific college *Galilée*, Tarvisio, Italy
Computer graphics teacher	1990-1991	Non-military national service, Udine, Italy
Master (*Tesi di Laurea*) in Computer Science	1990	University of Udine, Italy

Publications / Publications

Type	**#**	**Journals, conferences and workshops**
International journals with anonymous referee process (other journal-size papers are marked in bold of next item)	11	Transaction on Programming Languages ans Systems (TOPLAS), Annals of Pure and Applied Logics (APAL), Future Generation Computer Systems (FGCS), Journal of Logic and Computation (JLC), Information and Computation (IC), Theoretical Computer Science (TCS), Journal of Automated Reasoning (JAR), Mathematical Structures in Computer Systems (MSCS), Fundamenta Informaticae
International conferences and workshops with anonymous referee process	33	POPL, OOPSLA, **ECOOP**[2], MIT-JICSLP, CSL, MFCS, LFCS, FOSSACS, PPDP, FM, TAPSOFT/CAAP, LPAR, TYPES, ASIAN, I2CS, JVA, ARCS, IEEE-TOOLS, MERLIN, **WRLA**, **WRS**, WESTAPP, GULP-PRODE, ATSC, **TERMGRAPH**, ITRS, WWV, **DCM**
Nat. Conf./Workshops with anony. ref.	4	JFLA, SISFORM
International Workshops (no proc.)	4	RHO, TYPES, FUN, ATSC
Research Rep./Deliverables/Misc	6	INRIA, UNIUD, UNITO, IST-FP6
Software Patents and Reference Manuals	4	Momix©CSELT, {iRho,Snake}©INRIA
Unpublished/Submitted	9	TLCA
Thesis	3	Habilitation Thesis, Ph.D. Thesis, Master Thesis
Teaching Material/Course Notes	8	ENSMN, ENSL, Sussex University, Turin University

Enseignements / Teaching

Intitulé/Title[1]	Année Year	Étudiants Students	Niveau/Level : 1er=Bac+3 1er	2ème	3ème	DEA	Répartition/Kindof Cours	TD	Conf.
USUSX CoCo	03-04	15			×		15h	10h	
USUSX Fun	03-04	35	×				15h	10h	
DEA Math Nice	03-04	4				×	9h		
ENSMN SI 151	01-02	21			×		22h		13h
DEA IEAM Nancy	01-02	9				×	10h		
ENSMN SI 151	00-01	15			×		27h		18h
ENSMN SI 142	00-01	31		×			35h		10h
ENSMN TCS 23	00-01	14	×					61h	
ENSMN Proj+Stages	00-01	7		×	×			40h	
ENSMN SI 131	00-01	33		×			7h		
DEA IEAM Nancy	00-01	16				×	10h		
ENSMN SI 151	99-00	17			×		27h		18h
ENSMN SI 142	99-00	32		×			45h		
ENSMN SI 131	99-00	34		×			9h	8h	
ENSMN SI 153	99-00	17			×		6h		
ENSMN TCS 23	99-00	14	×					61h	
ENSMN Proj+Stages	99-00	11	×	×	×			47h	
ENSL POOGL	98-99	37	×	×			48h	32h	
ENSL RLC	98-99	14		×			6h	26h	
LSP	96-97	50			×		20h	30h	
LYCÉE	96-97	51					225h		
LYCÉE	91-92	80					231h		
FORM CONTINUE	91-92	30						160h	

[1] En **gras** les conférences de taille et «contenu» journal / In **bold** conferences with size and contents "journal-like".
[2] Voir mon CV pour les détails des cours / See my CV for full course details.

Appendix C

Bibliographie de Luigi / Luigi's Bibliography

Sauf mention contraire, toutes les publications sont dans des journaux, congrès, ateliers, «Internationaux, issus de sélections et comités de lecture avec actes». Mes articles sont disponibles en format Postscript/Pdf sur :

All publications, unless not clearly marked, are published in international journals, conferences, and workshops, with an anonymous refereee process. My papers are available in Postscript/Pdf on

`http://www-sop.inria.fr/mascotte/Luigi.Liquori/ME/Pub-Liquori.html`

Legenda:
(X%) = Taux Acceptation /Acceptation rate (si disponible / if available),
(I) = Invité / Invited, **(S) = Sélectionné / Selected**

C.1 Journaux Internationaux / International Journals

1. L. Liquori, A Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM Transaction on Programming Languages and Systems*, 24 pages, ACM Press, accepted, to appear, 2007.

2. **Invited** L. Liquori, A Spiwack. Extending FeatherTrait Java with Interfaces. In *Mario Coppo, Mariangiola Dezani and Simona Ronchi della Rocca Festschrift. Theoretical Computer Science*, 25 pages, Elsevier, accepted, to appear, 2007.

3. **Selected** R. Chand, M. Cosnard, L. Liquori. Powerful Resource Discovery for Arigatoni Overlay Network. *Future Generation Computer Systems*, 16 pages, Elsevier, accepted, to appear, 2007.

4. L. Liquori, B. Serpette. iRho: An Imperative Rewriting-calculus. *Mathematical Structures in Computer Science*, 34 pages, Cambridge University Press, accepted, to appear, 2007.

5. L. Liquori and S. Ronchi della Rocca. Intersection-Types *à la* Church. *Information and Computation*, 24 pages, accepted, to appear, Elsevier, 2007.

6. **Invited** F. Honsell, M. Lenisa, L. Liquori. A Framework for Defining Logical Frameworks. In *Gordon D. Plotkin Festschrift. Electronic Notes in Theoretical Computer Science*, 172, pages 399-436, Elsevier, 2007.

7. A. Ciaffaglione, L. Liquori, M. Miculan. Reasoning about Object-based Calculi in (Co)Inductive Type Theory and the Theory of Contexts. *Journal of Automated Reasoning*, 54 pages, Kluwer Academic Publishers, accepted, to appear, 2007.

8. P. Lescanne, L. Liquori, D. Dougherty. Addressed Term Rewriting Systems: Application to a Typed Object Calculus. *Mathematical Structures in Computer Science*, 16(4), pages 667-709, Cambridge University Press, 2006.

9. M. Bugliesi, G. Delzanno, L. Liquori and M. Martelli. Object Calculi in Linear Logic. *Journal of Logic and Computation*, 10(1), pages 75–104, Oxford University Press, 2000.

10. V. Bono, M. Bugliesi, M. Dezani and L. Liquori. A Subtyping for Extendible, Incomplete Objects. *Fundamenta Informaticae*, 38(4), pages 325–364, IOS Press, 1999.

11. S. van Bakel, L. Liquori, S. Ronchi della Rocca and P. Urzyczyn. Comparing Cubes of Typed and Type Assignment Systems. *Annals of Pure and Applied Logic*, 86(3), pages 267–303, Elsevier / North Holland, 1997.

C.2 Conférences et ateliers internationaux / International Conferences and Workshops

Note. J'ai décidé de mettre 6 articles de journaux dans cette section. Les articles publiés dans DCM'06, TERMGRAPH'04, WRLA'04 et WRS'03 sont publiés en Electronic Notes in Computer Science ; ce sont de vrais articles de «taille» et de «contenu» journal. La conférence internationale ECOOP-97-98 accepte des versions finales de 25 pages; elle est *de facto* considérée comme un journal.

Note. I have decided to put 6 "journal-size" papers in this section. The DCM'06, TERMGRAPH'04, WRLA'04, and WRS'03 papers are published in Electronic Notes in Computer Science, but they are definitively journal versions. ECOOP-97-98 international conference accepts 25 pages long camera ready; it is *de-facto* considered as a journal.

12. 24% R. Chand, M. Cosnard, L. Liquori. Improving Resource Discovery in the Arigatoni Overlay Network. In Proc. of *ARCS'07: 20th International Conference on Architecture of Computing Systems System Aspects in Pervasive and Organic Computing*, Zurich, Switzerland. *Lecture Notes in Computer Science* 4415, pages 98–111, Springer Verlag, 2007.

13. 46% D. Benza, M. Cosnard, L. Liquori, M. Vesin. Arigatoni: A Simple Programmable Overlay Network. In Proc. of *JVA'06, John Vincent Atanasoff International Symposium on Modern Computing*, Sofia, Bulgaria, pages 82–91, IEEE Computer Society, 2006.

14. 32% R. Chand, M. Cosnard, L. Liquori. Resource Discovery in the Arigatoni Overlay Network. In Proc. of *I2CS'06, International Workshop on Innovative Internet Community Systems*, Neuchatel, Switzerland. *Lecture Notes in Computer Science*, 13 pages, to appear, Springer Verlag, 2007.

15. 53% M. Cosnard, L. Liquori, R. Chand. Virtual Organizations in Arigatoni. In Proc. of *DCM'06, 3rd International Workshop on Developments in Computational Models*, Venice, Italy. *Electronic Notes in Theoretical Computer Science*, 171(3), pages 55–75, Elsevier, 2007. 22 pages = journal version

16. L. Liquori and F. Honsell, and R. Redamalla. A Language for Verification and Manipulation of Web Documents: (Extended Abstract). In Proc. of *WWV'05, 1st International Workshop on Automated Specification and Verification of Web Sites*, Valencia, Spain. *Electronic Notes in Theoretical Computer Science*, 157(2), pages 67–78, Elsevier, 2006.

17. 55% L. Liquori. iRho: the Software (System Description). In Proc. of *DCM'05, 2nd International Workshop on Developments in Computational Models*, Lisbon, Portugal. *Electronic Notes in Theoretical Computer Science*, 135(3), pages 85–96, Elsevier, 2006.

18. L. Liquori, S. Ronchi della Rocca. Towards an Intersection Typed System à la Church. In Proc. of *ITRS'04, Workshop on Intersection Types and Related Systems. Electronic Notes in Theoretical Computer Science*, 136, pages 43–56, Elsevier, 2005.

19. D. Dougherty, P. Lescanne, L. Liquori, F. Lang. Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics: Extended Abstract. In Proc. of *TERMGRAPH'04, 2nd International Workshop on Term Graph Rewriting. Electronic Notes in Theoretical Computer Science*, 127(5), pages 57–82, Elsevier, 2005. 25 pages = journal version

20. 47% L. Liquori, B. P. Serpette. An Imperative Rewriting Calculus. In Proc. of *PPDP'04, 6th ACM SIGPLAN Conference on Principle and Practice of Declarative Programming.* Verona, Italy, pages 167–179, The ACM Press, 2004.

21. L. Liquori and B. Wack. The Polymorphic Rewriting Calculus [Type Checking *vs.* Type Inference]. In Proc. of *WRLA'04, 6th International Workshop on Rewriting Systems and Applications.* Barcelona, Spain. Volume 117, pages 89-111 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2005. 22 pages = journal version

22. **Invited** H. Cirstea, C. Kirchner, L. Liquori, B. Wack. Rewrite Strategies in the Rewriting Calculus. In Proc. of *WRS'03, 3rd International Workshop on Reduction Strategies in Rewriting and Programming*, Valencia, Spain. *Electronic Notes in Theoretical Computer Science*, 86(4), pages 593-624, Elsevier, 2005. 31 pages = journal version

23. A. Ciaffaglione, L. Liquori, M. Miculan. Reasoning on an Imperative Object-based Calculus in Higher Order Abstract Syntax. In Proc. of *MERLIN'03, 2nd ACM SIGPLAN Workshop on MEchanized Reasoning about Languages with varIable biNding.* Uppsala, Sweden. The ACM Digital Library, 2003.

24. **Selected** H. Cirstea, L. Liquori, B. Wack: Rewriting Calculus with Fixpoints: Untyped and First-Order Systems. In Proc. of *Types '03, International Workshop on Types for Proof and Programs.* Turin, Italy. *Lecture Notes in Computer Science* 3085, pages 147-161, Springer Verlag, 2003.

25. 41% A. Ciaffaglione, L. Liquori, M. Miculan. Imperative Object-based Calculi in (Co)Inductive Type Theories. In Proc. of *LPAR'03, 10th International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Almaty, Kazakhstan. *Lecture Notes in Computer Science* 2850, pages 59–77, Springer Verlag, 2003.

26. 35% G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In Proc. of *POPL'03, 30th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages.* New Orleans, LA, USA, pages 250–261, The ACM Press, 2003.

27. **Invited** H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In Proc. of *WRLA'02, 4th International Workshop on Rewriting Systems and Applications.* Pisa, Italy. Volume 71 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2002.

28. H. Cirstea, C. Kirchner, L. Liquori and B. Wack. The Rho Cube: Some Results, Some Problems. In Proc. of *HOR'02, 1th International Workshop on Higher-Order Rewriting*, Copenhagen, Denmark, Electronic proceedings in `http://hor.pps.jussieu.fr/02/proc/proc.ps`, 2002.

29. 39% H. Cirstea and C. Kirchner and L. Liquori. The Rho Cube. In Proc. of *FOSSACS'01, 4th International Conference on Foundations of Software Science and Computation Structures.* Genova, Italy. *Lecture Notes in Computer Science* 2051, pages 77–92, Springer Verlag, 2001.

30. 28% H. Cirstea and C. Kirchner and L. Liquori. Matching Power. In Proc. of *RTA'01, 12th International Conference on Rewriting Techniques and Applications*. Utrecht, The Netherlands. *Lecture Notes in Computer Science* 2030, pages 168–183, Springer Verlag, 2001.

31. F. Lang, P. Lescanne, L. Liquori, D. Dougherty, and K. Rose. A Generic Object-Calculus Based on Addressed Term Rewriting Systems (extended abstract). In Proc. of *WEST-APP'01, 4th International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs*. Utrecht, The Netherlands. *Logic Group Preprint series* No 210, pages 6–25. Printed by the University of Utrecht, 2001.

32. 64% D. Colnet, L. Liquori. Match-O, a Statically Safe (?) Dialect of Eiffel. In Proc. of *IEEE-TOOLS '00, 37th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, Australia. pages 190–201, IEEE Computer Society, 2000.

33. 35% F. Lang, P. Lescanne, L. Liquori. A Framework for Defining Object-Calculi. In Proc. of *FM'99, World Congress on Formal Methods in the Development of Computing Systems*. Toulouse, France. *Lecture Notes in Computer Science* 1709, pages 963–982, Springer Verlag, 1999.

34. **Selected** L. Liquori. Bounded Polymorphism for Extensible Objects. In Proc. of *Types'98, International Workshop on Types for Proof and Programs*. Kloster Irsee, Germany. *Lecture Notes in Computer Science* 1657, pages 149–163. Springer Verlag, 1999.

35. 18% P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-Inflicted Extension. In Proc. of *ACM-SIGPLAN OOPSLA'98, International Symposium on Object Oriented, Programming, System, Languages and Applications*. Vancouver, Canada. 33(10), pages 166–178, ACM Press, 1998.

36. 19% L. Liquori. On Object Extension. In Proc. of *ECOOP'98, 12th European Conference on Object Oriented Programming*, Brussels, Belgium. *Lecture Notes in Computer Science* 1445, pages 498-552, Springer-Verlag, 1998. 25 pages = journal version

37. 19% L. Liquori. An Extended Theory of Primitive Objects: First Order System. In Proc. of *ECOOP'97, 11th European Conference on Object Oriented Programming*, Jyvaskyla, Finland. *Lecture Notes in Computer Science* 1241, pages 146–167, Springer-Verlag, 1997. 25 pages = journal version

38. 38% V. Bono, M. Bugliesi, M. Dezani and L. Liquori. Subtyping Constraints for Incomplete Objects. In Proc. of *TAPSOFT'97, 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, Lille, France. *Lecture Notes in Computer Science* 1214, pages 465–477, Springer-Verlag, 1997.

39. 18% L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In Proc. of *ASIAN'96, 2nd International Conference on Concurrency and Parallelism, Programming, Networking, and Security*, Singapore. *Lecture Notes in Computer Science* 1212, pages 129–141, Springer-Verlag, 1996.

40. 28% M. Bugliesi, G. Delzanno, L. Liquori and M. Martelli. A Linear Logic Calculus of Objects. In Proc. of *JICSLP'96, Joint International Conference and Symposium on Logic Programming*, Bonn, Germany. pages 79–94, The MIT Press, 1996.

41. 38% V. Bono, M. Bugliesi and L. Liquori. A Lambda Calculus of Incomplete Objects. In Proc. of *MFCS'96, 21st International Symposium of Mathematical Foundation of Computer Science, 1996*, Cracovie, Poland. *Lecture Notes in Computer Science* 1113, pages 218–229, Springer-Verlag, 1996.

42. 25% V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In Proc. of *CSL'94, 8th International Conference of Computer Science Logic*, Kazimierz, Poland. *Lecture Notes in Computer Science* 933, pages 16–30, Springer-Verlag, 1995.

43. S. van Bakel, L. Liquori, S. Ronchi della Rocca and P. Urzyczyn. Comparing Cubes. In Proc. of *LFCS'94. 3rd International Symposium on Logical Foundations of Computer Science*, St. Petersbourg, Russia. *Lecture Notes in Computer Science* 813, pages 353–365, Springer-Verlag, 1994.

44. L. Liquori and M.L. Sapino. Dealing with Explicit Exceptions, In Proc. of *Joint Conference on Declarative Programming, Gulp-Prode'94*, Peniscula, Spain. SPUPV-94.2046, pages 296–308, Printed by the University of Valencia, 1994.

C.3 Conférences et ateliers nationaux / National Conferences and Workshops

45. H. Cirstea, C. Kirchner, L. Liquori, B. Wack. Polymorphic Type Inference for the Rewriting Calculus. In Proc. of *JFLA'06, 17ème Journées Francophones des Langages Applicatifs*, Pauillac, France, pages 57–69, Édition INRIA, 2006.

46. 63% P. Di Gianantonio, L. Liquori and F. Honsell. A Lambda Calculus of Objects with Self-Inflicted Extension (extended abstract). In Proc. of *SisForm'98, 1st Workshop sui Sistemi Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione del Software*, Rome, Italy, SI-98/11, pages 39–40, Printed by the University of Rome I *La Sapienza*, 1998.

47. L. Liquori. Diamond Types for Extendible Objects. At *FUN'98, Fourth Italian Workshop on Functional Programming*, Como, Italy, 1998.

48. L. Liquori and M.L. Sapino. Operational and Denotational Semantics for a Logic Language with explicit Exceptions. In *FUN'95, First Italian Workshop on Functional Programming*, Bologna, Italy, 1995.

C.4 Ateliers internationaux (sans acte) / International Workshops (no proceeding)

49. F. Honsell, M. Lenisa, L. Liquori. A Framework for Defining Logical Frameworks. At *3rd Workshop on the Rewriting Calculus*, Kings College, London, UK. `http://rho.loria.fr/workshop2006.html`, 2006.

50. L. Liquori. Some Progress in the SW iRho. At *2nd Workshop on the Rewriting Calculus*, LIX, Paris, France, `http://rho.loria.fr/workshop2005.html`, 2005.

51. L. Liquori and A. Spiwack. OhML. (The "Zen Essence" of Methods-as-Functions). At *Types'04, International Workshop on Types for Proof and Programs.* LIX Palaiseau, Paris, France, `http://types2004.lri.fr/index.html`, 2004.

52. L. Liquori. A Typed Axiomatic Object Calculus with Subtyping. At *ATSC'95, Workshop on Advances In Type Systems For Computing*, Newton Institute, Cambridge, UK. `http://www.seas.upenn.edu/~sweirich/types/archive/1996/msg00006.html`, 1995.

C.5 Rapports de recherche non publiés, divers / Research Reports not Published Elsewhere, Deliverables, Miscellaneous

53. H. Cirstea and E. Coquery and W. Drabent and F. Fages and C. Kirchner and L. Liquori and B. Wack and A. Wilk. Types for REWERSE reasononing and query language. REWERSE Network of Excellence. Deliverable I3-D4, 2005.

54. L. Liquori. Book Review: Formal Methods for Open Object-Based Distributed Systems. *The Computer Journal*, 46(6). Oxford University Press - British Computer Society, 2003.

55. L. Liquori & the Miró Team. Miró: Systèmes à Objets, Types et Prototypes : Sémantique et Validation, Proposition de Projet INRIA, V1.3, 58 pages, in French, `http://www-sop.inria.fr/interne/vie/comites/cp/AvantProjets/Miro/MiroV1.3.ps.gz`, 2002.

56. L. Liquori & the Miró Team. Miró: Systèmes à Objets, Types et Prototypes : Sémantique et Validation, Rapport d'activité INRIA, `http://www.inria.fr/rapportsactivite/RA2001/miro/miro_tf.html`, 2001-2002-2003.

57. L. Liquori. Playing with Diamonds. Research Report, Dipartimento di Matematica ed Informatica, University of Udine, Italy, 1998.

58. L. Liquori. An Extended Theory of Primitive Objects : First and Second Order Systems. Research Report CS-23-96, Dipartimento di Matematica ed Informatica, University of Turin, Italy, 1996.

C.6 Logiciels et manuels de référence / Software and Reference Manuals

59. L. Liquori. Snake: The First Ascii-Oriented Script Language Based on the Original Screenplay of the Imperative Rewriting Calculus V1.1, `http://www-sop.inria.fr/mascotte/Luigi.Liquori/Snake`, 2005.

60. L. Liquori and B. Serpette. iRHO: interpreter and type checker, and proof certification. `http://www-sop.inria.fr/mascotte/Luigi.Liquori/iRho`, 2004.

61. D. Bacchiega, D. Gotta, L. Liquori, M. Porta and M. Ramella Votta. *Ipotesi evolutive del Sistema MOMIX per la Specifica e Qualificazione di Sistemi TMN*. In Italian. Technical Report DTR 98.0089, 27 pages, CSELT, Centro Studi e Laboratori Telecomunicazioni (now Telecom Italia Labs), Turin, Italy, 1998.

62. D. Bacchiega, D. Gotta, L. Liquori and R. Rossi. *MOMIX-emSDH2. Simulatore di Agent di Element Manager SDH2.0. Versione 2.2.3. Manuale Utente e Guida di Riferimento.* In Italian. Technical Report DTR 97.0413, 99 pages, CSELT, Centro Studi e Laboratori Telecomunicazioni (now Telecom Italia Labs), Turin, Italy, 1997.

C.7 Non publiés, soumis /Unpublished, Submitted

63. F. Honsell, M. Lenisa, L. Liquori, I. Scagnetto. Constraint Logical Framework. Submitted, 2007.

64. F. Honsell, M. Lenisa, L. Liquori. An Axiomatic Logical Framework, in preparation, 2007.

65. C. Bertolissi, H. Cirstea, G. Faure, C. Kirchner, L. Liquori, and B. Wack. The Rewriting Calculus. Book, in preparation, 2007.

66. L. Liquori. On Demand Type Systems (How to plug *your* own type system in *our* virtual machine!), 2006.

67. S. Fechter, and L. Liquori, and T. Hardin. Mini-Foc: A Kernel Calculus for Certified Computer Algebra [Ongoing Work], 2006.

68. L. Liquori, and A. Spiwack. An Object-Oriented Logical Framework [Preliminary Report], 2005.

69. L. Liquori, and A. Spiwack. OhML. (The "Zen Essence" of Methods-as-Functions), 2005.

70. A. Ciaffaglione, P. Di Gianantonio, L. Liquori, and F. Honsell. Foundations for Dynamic Objet Reclassifications, Extended version of OOPSLA'98, 2005.

71. L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. Extended version of ASIAN'96, 2000.

72. L. Liquori. The Deep Blue Calculus. Manuscript, 1998.

C.8 Thèses / Thesis

73. L. Liquori. `this.Peter()`, a.k.a. *Peter, le langage qui n'existe pas... (Peter, the language that does not exist...)*, 2007[1]

74. L. Liquori. *Type Assigment Systems for Lambda Calculi and for the Lambda Calculus of Objects*. Ph.D. Thesis, 193 pages, in English and Italian, Department of Computer Science, University of Turin, Italy, 1996.

75. L. Liquori. *Semantica e Pragmatica di un Linguaggio Funzionale con le Continuazioni Esplicite*, Ms.D. Thesis, 74 pages, in Italian, Department of Mathematics and Computer Science, University of Udine, Italy, 1990.

C.9 Matériel de cours / Teaching Material

76. L. Liquori. Functional Programming. G5031, Course Slides, University of Sussex, Brighton, UK, 2004.

77. L. Liquori. Computability and Complexity. G5003, Course Slides, University of Sussex, Brighton, UK, 2004.

78. L. Liquori. A Brief Introduction to Corba. Course Notes SI 131: *Conception d'architectures logicielles*. École des Mines de Nancy, France, 1999-2000.

79. L. Liquori. On Object Calculi. Course Notes SI 142: *Fondements de l'algorithmique et de la programmation*. École des Mines de Nancy, France, 1999-2000.

80. L. Liquori. Les Langages ASN.1 et GDMO. Course Notes *SI 151 : Réseaux et télécommunications*. École des Mines de Nancy, France, 1999-2000.

81. L. Liquori. Le Bug de Java. Course Notes SI 153: *Sûreté des systèmes informatiques*. École des Mines de Nancy, France 1999-2000.

82. L. Liquori and K. Tombre. On C++ Compilers. Course Notes SI 131: *Conception d'architectures logicielles*. École des Mines de Nancy, France, 1999-2000.

83. L. Liquori. An Introduction to SmallTalk. Course Notes *POOGL: Programmation à Objets et Génie Logiciel*, École Normale Supérieure de Lyon, France, 1998-1999.

[1] Desolé pour la référence récursive / Sorry for the recursive-reference.

Bibliography

[Abr96] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[ACL+07] E. Allen, D. Chase, V. Luchangco, J-W Maessen, G.L. Steele S. Ryu, and S. Tobin-Hochstadt. The Fortress Language Specification, version 0.618., 2007. `http://research.sun.com/projects/plrg/fortress0618.pdf`.

[AHMP92] A. Avron, F. Honsell, I.A. Mason, and R. Pollack. Using Typed Lambda Calculus to Implement Formal Systems on a Machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992.

[Bar92] H. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov.M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[BDKT03] K. B. Bruce, R. L. Scot Drysdale, C. Kelemen, and A. B. Tucker. Why math? *Commun. ACM*, 46(9):40–44, 2003.

[BG95] E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In *Proc. of ECOOP*, LNCS, pages 102–126. Springer Verlag, 1995.

[Bra04] G. Bracha. Keynote address: Towards Secure Systems Programming Languages. In *SAC*, pages 1–2, 2004. `http://pico.vub.ac.be/~wdmeuter/RDL04/papers/Bracha.pdf`.

[Bru99] K. B. Bruce. Formal Semantics and Interpreters in a Principles of Programming Languages Course. In *SIGCSE*, pages 331–335, 1999.

[Bru02] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002. `http://www.cs.williams.edu/~kim/FOOLbook.html`.

[Car04] L. Cardelli. Type Systems. In A.B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.

[CH84] M. Cresswell and G. Hughes. *A Companion to Modal Logic*. Methuen, 1984.

[CH05] E. Chailloux and G. Henry. The O'Jacare Home Page, 2005. `http://www.pps.jussieu.fr/~henry/ojacare/download.en.html`.

[Chu41] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1941.

[dB80] N. G. de Bruijn. A Survey of the Project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[DDDCG01] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle : Dynamic object re-classification. In *ECOOP*, pages 130–149, 2001.

[DDDCG02] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More Dynamic Object Re-classification: FickleII. *ACM TOPLAS*, 24(2):153–191, 2002.

[DNS+06] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A Mechanism for Fine-grained Reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[Dow03] G. Dowek. *Au coeur d'une calculatrice*. Le Pommier, 2003.

[Dow07] G. Dowek. *Les Métamorphoses du calcul : Une étonnante histoire des mathématiques*. Le Pommier, 2007.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FR04] K. Fisher and J. Reppy. Statically Typed Traits. `http://www.cs.uchicago.edu/files/tr_authentic/TR-2003-13.pdf`. The early version *"A Typed Calculus of Traits"* has been presented at FOOL 10, 2004.

[HHP94] R. Harper, F. Honsell, and G. D. Plotkin. A Framework for Defining Logics. *Journal of ACM*, 40(1):143–184, 1994.

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. thèse, Université de Paris 7 (France), 1976.

[IPW01] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[KBN99] F. Kamareddine, R. Bloo, and R. Nederpelt. On π-conversion on the λ-cube and the Combination with Abbreviations. *Annals of Pure and Applied Logics*, 97(1-3):27–45, 1999.

[KKM07] R. Kopetz, C. Kirchner, and P.E. Moreau. Anti Pattern-Matching. In *Proc. of ESOP*, volume LNCS. Springer, 2007. To appear.

[Ler06] X. Leroy. The CamlJava Home Page, 2006. `http://caml.inria.fr/distrib/bazar-ocaml/camljava-0.3.tar.gz`.

[Log07] The LogiCal Team. The Coq Home Page, 2007. `http://coq.inria.fr/`.

[Pie02] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pie05] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[Plo75] G. Plotkin. Call by Name, Call by Value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[RBC+05] E. S. Roberts, K. B. Bruce, R. Cutler, J. H. Cross II, S. B. Grissom, K. Klee, S. Rodger, F. Trees, I. Utting, and F. Yellin. The ACM Java Task Force: Status Report. In *SIGCSE*, pages 46–47, 2005.

[Sca07] The Scala Team. The Scala Home Page, 2007. `http://scala.epfl.ch/`.

[SD05] C. Smith and S. Drossopoulou. Chai: Typed Traits in Java. In *Proc. of ECOOP*, volume 3586 of *LNCS*, pages 453–478. Springer Verlag, 2005.

[TKB01] A. B. Tucker, C. Kelemen, and K. B. Bruce. Our Curriculum has Become Math-Phobic! In *SIGCSE*, pages 243–247, 2001.